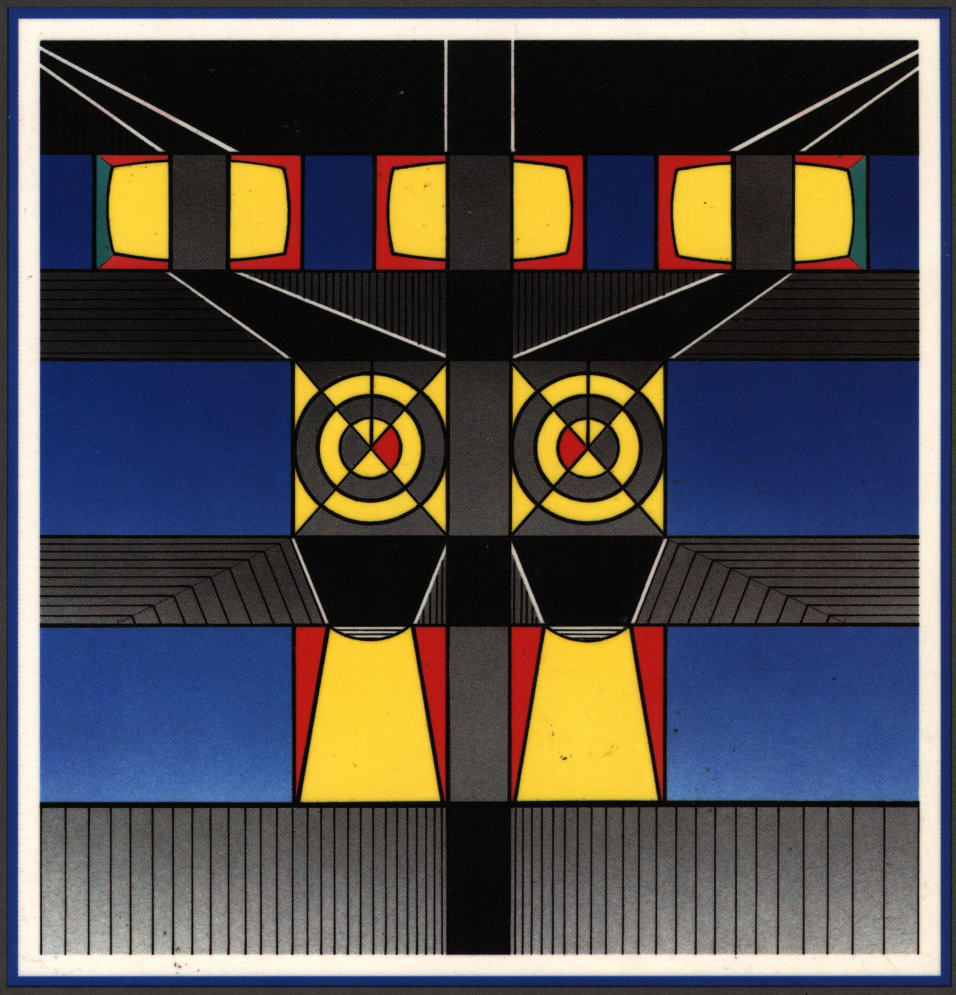
# Gegevensbanken

## Een inleiding

H.M. Blanken en C.J. Date

ACADEMIC SERVICE ADDISON-WESLEY



Module HB.1

# GEGEVENSBANKEN
een inleiding

Boeken verschenen in de Academic Service / Addison-Wesley serie:

*Basiskennis bestuurlijke informatiekunde* - O'Leary & Williams
*Bestuurlijke informatieverzorging* - v.Swigchem
*Database, een inleiding* - Date
*Inleiding systeemanalyse en systeemontwerp* - Davis
*Theorie en praktijk van besturingssystemen* - Peterson & Silberschatz

# Gegevensbanken

## Een inleiding

H.M. Blanken en C.J. Date

ACADEMIC SERVICE ADDISON-WESLEY

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Blanken, H.M.

Gegevensbanken, een inleiding / H.M. Blanken, C.J. Date. -
Schoonhoven : Academic Service. - Ill.
Met lit. opg.
ISBN 90-6233-355-9
SISO 521.2 SVS 8.12.3 UDC 681.3 NUGI 852
Trefw.: databanken.

10 9 8 7 6 5 4 3 2 1

Uitgegeven door: Academic Service
Postbus 81
2800 AB Schoonhoven
Het boek is door de auteur camery-ready aangeleverd.
Druk: Krips Repro Meppel
Bindwerk: Meeuwis, Amsterdam

Copyright © 1988 Academic Service

ISBN 90 6233 355 9
NUGI 852

Niets uit deze uitgave mag worden verveelvoudigd en/of openbaar gemaakt door middel van druk, fotokopie, microfilm, geluidsband, elektronisch of op welke andere wijze ook en evenmin in een retrieval system worden opgeslagen zonder voorafgaande schriftelijke toestemming van de uitgever.

# Inhoud

**Voorwoord** ix

**Hoofdstuk 1**
**Basisbegrippen**
1.1 Kennis, Informatie en Gegevens 1
1.2 Bedrijfsgegevens 2
1.3 Bestandsaanpak 6
1.4 Wat is een Database Systeem? 12
1.5 Voordelen van Databases 14
1.6 Nadelen van Databases 17
Opgaven 19

**Hoofdstuk 2**
**De Architectuur van een Database Systeem**
2.1 Inleiding 20
2.2 De Drie Niveaus van de Architectuur 20
2.3 Het Externe Niveau 23
2.4 Het Conceptuele Niveau 26
2.5 Het Interne Niveau 27
2.6 Afbeeldingen 27
2.7 Slotopmerking 28
Opgaven 29

**Hoofdstuk 3**
**Data Modellen: een Inleiding**
3.1 Inleiding 30
3.2 Het Relationele Data Model 31
3.3 Het Hiërarchische Data Model 34
3.4 Het Netwerk Data Model 37
3.5 Hogere Niveau Operatoren 41
Opgaven 45

**Hoofdstuk 4**
**Relationele Databases**
4.1 Inleiding 47
4.2 Domeinen 47
4.3 Relaties 50
4.4 Sleutels 55
4.5 De Twee Integriteitsregels 58
Opgaven 59

**Hoofdstuk 5**
**Ontwerp van Relationele Databases I**
**Globaal**
5.1 Inleiding 62
5.2 Entiteiten en Associaties 62
5.3 Niet-Atomaire Kenmerken 66
5.4 Integriteit en Bijwerkoperaties 67
5.5 De Onvolledige Ontwerpmethode 69
5.6 Normalisatie 71
5.7 De Volledige Ontwerpmethode 73
Opgaven 73

**Hoofdstuk 6**
**Ontwerp van Relationele Databases II**
**Normalisatie**
6.1 Inleiding 75
6.2 Functionele Afhankelijkheden 76
6.3 Eerste, Tweede en Derde Normaalvorm 78
6.4 Boyce/Codd Normaalvorm 86
6.5 Goede en Slechte Decomposities 91
6.6 Samenvatting en Slotopmerkingen 95
Opgaven 97

**Hoofdstuk 7**
**Manipulatie van een Relationele Database**
7.1 Inleiding 100
7.2 Eenvoudige Vragen 102
7.3 Join Vragen 106
7.4 Ingebouwde Functies 111
7.5 Gevorderde Aspecten 115
7.6 Bijwerkopdrachten 123
7.7 Concluderende Opmerkingen 128
Opgaven 130

**Hoofdstuk 8**
**DBTG Databases**
8.1 Inleiding 137
8.2 Architectuur 139
8.3 De Setconstructie: Hiërarchische Voorbeelden 141
8.4 De Setconstructie: Netwerk Voorbeelden 146
8.5 Singuliere Sets 152
8.6 Een Voorbeeld Schema 153
8.7 Membership Class 158
8.8 Set Selection 162
8.9 Verschillen Tussen Subschema en Schema 164
8.10 Een Voorbeeld Subschema 165
Opgaven 166

**Hoofdstuk 9**
**Ontwerp van DBTG databases**
9.1 Inleiding 168
9.2 Entiteiten en Associaties 168
9.3 Niet-Atomaire Kenmerken 170
9.4 Integriteit en Bijwerkoperaties 171
Opgaven 174

**Hoofdstuk 10**
**Manipulatie van een DBTG Database**
10.1 Inleiding 175
10.2 Currency 176
10.3 Behandeling van Uitzonderlijke Situaties 180
10.4 GET 181
10.5 STORE 182
10.6 ERASE 183
10.7 MODIFY 185
10.8 CONNECT 186
10.9 DISCONNECT 187
10.10 RECONNECT 187
10.11 FIND 188
10.12 KEEP LISTS 195
10.13 READY en FINISH, COMMIT en ROLLBACK 197
10.14 Database-condities 198
Opgaven 200

**Hoofdstuk 11**
**Het Interne Niveau**
11.1 Inleiding 201
11.2 Toegang tot de Database: Een Overzicht 204
11.3 Paginaverzamelingen en Bestanden 209
11.4 Indexering 215
11.5 Hashing 223
11.6 Wijzerketens 228
11.7 Slotopmerkingen 229
Opgaven 230

**Antwoorden op Opgaven** 232

**Literatuur** 264

**Appendix A**
SUBSET SQL 266

**Appendix B**
Subset DDL 270

**Appendix C**
SUBSET DML 273

**Index** 276

# VOORWOORD

Dit boek gaat over het ontwerpen en gebruiken van databases. Er wordt vanuit gegaan dat kennis van een hogere programmeertaal (bij voorkeur PASCAL of COBOL), alsmede enige globale kennis van bestandsorganisatie en apparatuur aanwezig is.

Na een uiteenzetting over de verschillen tussen bestanden en databases, komt de ANSI/SPARC architectuur aan de orde. Het conceptuele, externe en interne niveau van database systemen wordt beschreven. Aan de hand van deze architectuur worden het relationele en het DBTG data model uitgelegd, zowel wat betreft de gegevensstructurering (DDL) als gegevensmanipulatie (DML); met name SQL wordt behandeld (alhoewel niet tot in alle details). Het ontwerpen van een conceptueel schema voor zowel het relationele als het DBTG model komt expliciet aan de orde. Hierbij wordt ook de rol van normalisatietechnieken besproken. Het laatste hoofdstuk behandelt het interne niveau van een database systeem, waarin aandacht wordt besteed aan het opslaan en terugzoeken van gegevens. Bijna ieder hoofdstuk heeft een verzameling opgaven (meestal met uitwerkingen) die de lezer in staat stellen de opgedane kennis te toetsen.

De invoering van begrippen gebeurt niet al te formeel en vaak aan de hand van vele voorbeelden. De inhoud is een bewerking van een tiental hoofdstukken uit het boek "An Introduction to Database Systems", volume 1 (derde en vierde druk) dat door C.J. Date is geschreven en door Addison-Wesley Publishing Company is uitgegeven. De stof dekt de eisen voor het HB1 module in de opleiding tot het AMBI diploma, maar daarnaast is het boek zeer zeker ook bedoeld voor gebruik in vele andere opleidingen in het hoger onderwijs, waarin een degelijke doch elementaire inleiding over database systemen vereist is.

Prof. dr. J. van Leeuwen, Dr. W. Wesselius, Drs. A. IJbema en A. Kemper hebben door hun commentaar een belangrijke bijgedrage aan de totstandkoming van dit boek geleverd, hetgeen vanzelfsprekend zeer op prijs wordt gesteld. Mevrouw M. Demmer heeft het nodige correctiewerk verricht, terwijl Marian Elfrink, Willem Witte en Jos Blanken de omslag en de figuren voor hun rekening namen. Al deze personen danken we zeer voor hun steun. Mochten er desondanks bij bestudering van de stof nog fouten of andere onvolkomenheden worden ontdekt, dan worden we daar graag op geattendeerd.

Juli 1988 H.M.B/C.J.D.

# 1
# Basisbegrippen

## 1.1 KENNIS, INFORMATIE EN GEGEVENS

In de huidige maatschappij speelt kennis een grote rol. Onder *kennis* wordt hier verstaan datgene wat men weet over een bepaald onderwerp of gebied. Kennis kan men verwerven door fenomenen uit dat gebied waar te nemen, over het waargenomene na te denken, maar ook door uit kennisbronnen te putten. In het laatste geval maakt men gebruik van door anderen vergaarde kennis waardoor er kennisoverdracht plaatsvindt. Kennisoverdracht eist het expliciet maken van kennis, hetgeen niet altijd goed mogelijk is (denk bijvoorbeeld aan de kennis die nodig is om een schaakstelling of de toestand van een patiënt te beoordelen). Kennis die overdraagbaar is noemen we *informatie* en het ontvangen van informatie zorgt in het algemeen voor kennisvergroting. Dit geldt zowel voor personen individueel als voor groepen van personen. Bijvoorbeeld, in een bedrijf werken vele personen samen en deze personen (de werknemers) hebben gemeenschappelijke doelen (het fabriceren van produkten, het maken van winst, e.d.). Om deze doelen goed te kunnen nastreven is informatie nodig. Het deel van de reële wereld dat relevant is voor het goed functioneren van het bedrijf, noemen we het *object systeem*. Zo zullen in het algemeen klanten, werknemers en artikelen voor een bedrijf relevante begrippen zijn en dus tot het object systeem behoren.

Het overdragen van informatie (kennis) gebeurt door middel van *communicatie*. Daartoe moet de informatie gerepresenteerd worden. De gerepresenteerde informatie noemen we *gegevens*. De representatie kan geschieden op een vluchtige en op een permanente manier. De eerste manier houdt in dat na de communicatie de representatie verdwenen is; voorbeelden hiervan zijn gebaren, gesprekken, gelaatsuitdrukkingen en dergelijke. De permanente representatie (beter is semi-permanent, omdat 'niets eeuwig' is), maakt het mogelijk dat de

gerepresenteerde informatie meermalen gebruikt en gecommuniceerd wordt. Bekende voorbeelden zijn boeken, foto's, brieven, tekeningen, etc. We zullen ons in dit boek beperken tot permanente representaties.

Aangezien het ondoenlijk is om alle relevante informatie te onthouden, kan permanent gerepresenteerde informatie een belangrijke rol spelen bij het terugwinnen van eenmaal vergaarde, doch inmiddels vergeten kennis. De verzameling van permanent gerepresenteerde informatie noemen we de gegevensverzameling. Op de gegevensverzameling moet een zogenaamde 'leesfunctie' gedefinieerd zijn, waarmee gewenste gegevens uit de verzameling opgevraagd kunnen worden. Natuurlijk moeten ook nieuwe gegevens aan de gegevensverzameling kunnen worden toegevoegd en moeten gegevens die niet meer belangrijk zijn, kunnen worden verwijderd.

Aan de bewerkingen op de gegevensverzameling kunnen vele eisen worden gesteld. Een belangrijke eis is dat de bewerkingen snel moeten kunnen worden uitgevoerd. Om aan deze eis te voldoen, worden vaak computers ingeschakeld en geschiedt de representatie van informatie in computergeheugens. Vanaf nu gaan we uit van het gebruik van computers bij het vervullen van de informatiebehoefte.

## 1.2 BEDRIJFSGEGEVENS

In de vorige paragraaf is het woord bedrijf al enige malen gevallen. 'Bedrijf' is een algemene term voor een willekeurige commerciële, technische, wetenschappelijke of andere organisatie. Voorbeelden zijn:

een fabriek,
een bank,
een ziekenhuis,
een universiteit,
een ministerie.

Ieder bedrijf moet veel gegevens met betrekking tot het functioneren bijhouden. Deze gegevens worden de "bedrijfsgegevens" genoemd. Bovengenoemde bedrijven zullen, respectievelijk, tenminste gegevens bijhouden over:

produkten,
rekeningen,
patiënten,
studenten,
planning.

**1.2.1 Entiteiten.** Laten we een fabriek eens nader beschouwen. Een dergelijk bedrijf zal informatie willen hebben over de *projecten* die momenteel worden uitgevoerd; de *artikelen* die in de projecten worden gebruikt; de *leveranciers* die deze artikelen kunnen leveren; de *magazijnen* waar de artikelen worden opgeslagen; de *werknemers* die in de projecten werken, etc. (zie Fig. 1.1). De cursief gedrukte woorden geven de entiteiten weer waarover informatie moet worden bijgehouden. Hierbij wordt 'entiteit' als volgt gedefinieerd:

Een *entiteit* is ieder onderscheidbaar object waarover informatie moet worden bijgehouden.

"Onderscheidbaar" houdt in dat de ene entiteit van de andere moet kunnen worden onderscheiden. Zo zouden mieren entiteiten kunnen zijn als we in staat waren om ze van elkaar te onderscheiden; kunnen we dat niet, dan beschouwen we een mier niet als een entiteit (en mogelijk een groep mieren wel, mits een groep mieren van andere groepen te onderscheiden is). Een entiteit kan zowel concreet (artikel, persoon) als abstract (religie, jazz) zijn.

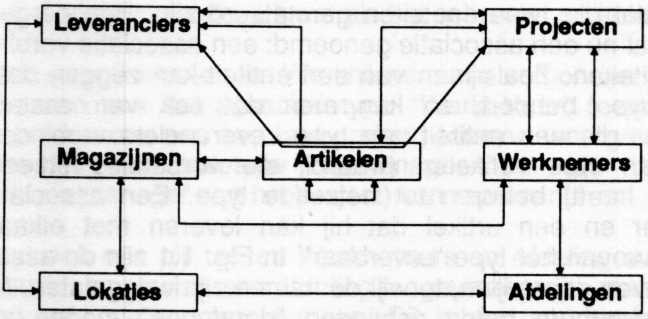



**Fig. 1.1 Een voorbeeld van bedrijfsgegevens.**

**1.2.2 Kenmerken.** We beschouwen nu de "informatie die men over entiteiten wil bijhouden" en nemen daarbij als voorbeeld de entiteit "artikel". Het kan zinvol zijn om van artikelen de naam, de beschrijving, het gewicht en de kleur bij te houden. Dit noemt men *kenmerken* van de betreffende entiteit. De genoemde artikel-kenmerken hebben in het algemeen *een* waarde: een artikel heeft één naam, één nummer, etc. Het kan echter ook voorkomen dat een kenmerk een *verzameling waarden* heeft. Stel dat artikelen in vele kleuren worden geleverd en dat deze kleuren van het specifieke artikel afhangen (wat vaak het geval is). Dan zou voor ieder artikel de waarde van het kenmerk 'kleur' een verzameling van de van toepassing zijnde kleuren zijn.

Entiteiten behoren tot *hetzelfde type* als we van de entiteiten dezelfde kenmerken bijhouden. Het type van een entiteit wordt dus gedefinieerd door de kenmerken die we bijhouden. Zo behoren alle artikelen tot het *entiteittype* 'artikelen', omdat we van artikelen dezelfde kenmerken (naam, beschrijving, etc.) willen bijhouden. Het verschil tussen *type* en *exemplaar* (in het Engels *occurrence*, letterlijk vóórkomen) is zeer wezenlijk. Bij een type hoort een verzameling exemplaren van dat type; zo hoort bij het entiteittype 'artikelen' de verzameling van de artikelen. In Fig. 1.1 zijn entiteittypen door rechthoeken weergegeven. We zullen vanaf nu de naam van een entiteittype met een hoofdletter schrijven, bijvoorbeeld Artikelen.

**1.2.3 Associaties.** In Fig. 1.1 verbindt een (twee-puntige) pijl de entiteittypen Leveranciers en Artikelen. Deze pijl duidt aan dat iedere leverancier bepaalde artikelen kan leveren en dat omgekeerd, ieder artikel door bepaalde leveranciers kan worden geleverd (het is beter om over 'soort artikel' in plaats van 'artikel' te spreken; als men immers zegt dat een vliegtuigfabrikant een vliegtuig kan leveren, dan wordt in het algemeen een soort vliegtuig bedoeld en niet een specifiek exemplaar).

Het feit dat een zekere leverancier een bepaald artikel kan leveren, houdt in dat die twee entiteiten gerelateerd zijn. Een dergelijke relatie wordt vanaf nu een associatie genoemd: een associatie verbindt twee (of meer) entiteiten. Zoals men van een entiteit kan zeggen dat zij tot een bepaald *type* behoort, zo kan men dat ook van associaties: alle associaties die een entiteit van type Leveranciers verbinden met een entiteit van type Artikelen (waarbij die verbinding steeds dezelfde betekenis heeft) behoren tot hetzelfde type. Een associatie die een leverancier en een artikel dat hij kan leveren met elkaar verbindt, noemen we van het type 'Leverbaar'. In Fig. 1.1 zijn de associatietypen weergegeven door pijlen, terwijl de namen zijn weggelaten. Merk op dat associatietypen in beide richtingen "doorlopen" mogen worden. Zo kunnen associaties van type 'Leverbaar' worden gebruikt bij het beantwoorden van de volgende twee vragen: (a) "Welke artikelen kunnen door een bepaalde leverancier geleverd worden?" en (b) "Welke leveranciers kunnen een bepaald artikel leveren?".

Een belangrijk punt is dat *associaties* net zo goed *tot de bedrijfsgegevens behoren* als de geassocieerde entiteiten. Daarom moeten ze ook in de gegevensverzameling gerepresenteerd worden. We zullen elders in dit boek nog verscheidene manieren laten zien om dit te doen.

Fig. 1.1 illustreert nog een aantal andere punten:

1. Hoewel de meeste associatietypen in het diagram twee entiteittypen verbinden, hoeft dit niet altijd het geval te zijn. In het voorbeeld is een pijl aanwezig die drie rechthoekjes verbindt, namelijk die van Leveranciers, Artikelen en Projecten. Dit betekent, dat bepaalde leveranciers bepaalde

artikelen leveren aan bepaalde projecten. Dit is in het algemeen *niet* hetzelfde als de combinatie van het associatietype tussen Leveranciers en Artikelen en dat tussen Artikelen en Projecten. Een voorbeeld: het gegeven "leverancier L2 levert artikel A4 aan project P3" geeft ons *meer* informatie dan "leverancier L2 levert artikel A4" en "artikel A4 wordt gebruikt in project P3" apart. We kunnen de bewering vervat in het eerste gegeven niet concluderen uit de tweede en de derde, terwijl het omgekeerde wel het geval is: als we weten dat de eerste bewering geldt dan gelden ook de tweede en de derde. Nog explicieter, als we weten dat L2 artikel A4 levert en dat A4 wordt gebruikt in P3, dan kunnen we concluderen dat L2 artikel A4 levert aan een zeker project Px en dat een zekere leverancier Ly artikel A4 levert aan P3, maar we kunnen niet concluderen dat Px gelijk is aan P3 of dat Ly gelijk is aan L2. Foutieve gevolgtrekkingen zoals hier aangegeven, zijn voorbeelden van wat soms wordt aangeduid als de *verbindingsval* (connection trap).

2. Het voorbeeld toont ook een pijl, die alleen maar betrekking heeft op de rechthoek Artikelen. Deze pijl representeert een associatietype tussen Artikelen en Artikelen, namelijk het associatietype dat bepaalt dat sommige artikelen componenten van andere artikelen (met een complexere structuur) zijn. Zo kan een schroef een component van een scharnier zijn, terwijl zowel de schroef als het scharnier artikelen zijn.

3. De entiteittypen Projecten en Werknemers zijn verbonden door twee pijlen. Een ervan zou het associatietype "Werkt-Voor" (een bepaalde werknemer werkt voor een bepaald project) kunnen weergeven, terwijl de andere het associatietype "Is-Manager-Van" (de betreffende werknemer is de manager van het project) kan representeren.

4. Er bestaat ook een pijl tussen Werknemers en Afdelingen. Vaak zal een werknemer aan slechts *een* afdeling toegewezen zijn, maar zullen in een afdeling vele werknemers werken. Dit is een voorbeeld van een *veel-op-een associatietype*, kortheidshalve soms weergegeven als n:1 associatietype. Een speciaal geval is het *een-op-een associatietype* (ook wel 1:1 associatietype). Een voorbeeld van dit laatste is het associatietype HUWELIJK dat een persoon van het mannelijk geslacht verbindt met een persoon van het vrouwelijk geslacht (zie Fig. 1.2).

5. Het associatietype dat Leveranciers en Artikelen verbindt heet een *veel-op-veel associatietype*, omdat in het algemeen een leverancier vele artikelen kan leveren en een artikel door vele leveranciers geleverd kan worden (zie ook Fig. 1.2). Let wel, een veel-op-veel associatietype houdt niet in dat elke leverancier meer dan één artikel *moet* leveren; het veel-op-veel associatietype houdt in dat een leverancier nul, een of meer artikelen kan leveren. Een veel-op-veel associatietype wordt ook wel een n:m associatietype genoemd.

Opmerking.
Vaak worden entiteiten en associaties als fundamenteel verschillend beschouwd. Echter, een associatie tussen entiteiten kan zelf ook weer als een entiteit worden beschouwd. Een entiteit is immers gedefinieerd als "een onderscheidbaar object waarover informatie moet worden bijgehouden". "Artikel A4 wordt opgeslagen in magazijn M8" is een object waarover we wellicht informatie willen bijhouden, bijvoorbeeld de opgeslagen hoeveelheid. Als zodanig is de associatie dan een entiteit. In met name [CHEN76], [CODD79], [DATE83] en [TSLO82] wordt dieper op deze materie ingegaan.

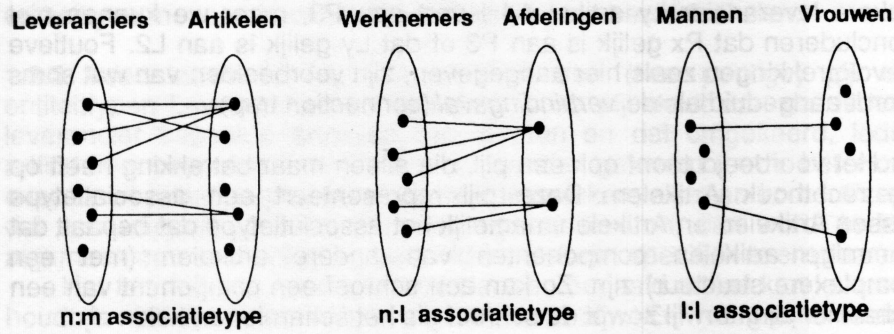



**Fig. 1.2 Voorbeelden van Associatietypen.**

## 1.3 BESTANDSAANPAK

We beschouwen nu de taak om bedrijfsgegevens in een computer op te slaan. Hiertoe voeren we, aan de hand van het entiteittype Werknemers, een drietal termen in:

- Per werknemer hebben we een aantal kenmerken, zoals naam, werknemernummer, adres en geboortedatum. Met ieder kenmerk correspondeert een opgeslagen veld. Een *opgeslagen veld* is de kleinste eenheid van opgeslagen gegevens die nog een eigen naam heeft gekregen. De waarde die het kenmerk werknemernummer van een zekere werknemer aanneemt, wordt in een veld van type WERKNO opgeslagen. (De woorden die een rol spelen bij de hierna te geven bestandsdefinitie worden met hoofdletters aangegeven.) Er zijn even veel exemplaren van het veldtype WERKNO als er werknemers zijn.
- Een *opgeslagen record* is een benoemde verzameling van gerelateerde opgeslagen velden. Een opgeslagen record in het WERKNEMERS bestand (voor het begrip bestand: zie hierna) zal bestaan uit een exemplaar van de opgeslagen veldtypen: WERKNO, NAAM, ADRES en GEBDATUM, in deze volgorde.

- Een *opgeslagen bestand* is de benoemde verzameling van alle opgeslagen records van een bepaald type. (We negeren de mogelijkheid dat een opgeslagen bestand records van meer dan een type bevat. Deze aanname heeft geen essentiële gevolgen voor de nu volgende uiteenzetting.)

De definitie van het bestand dat alle WERKNEMER records bevat, zou er (sterk schematisch) als volgt uit kunnen zien:

```
WERKNEMERS BESTAND.
WERKNEMER RECORD
        LENGTE = 70 CHARACTERS.
        WERKNO          TYPE = CHARACTER 8.
        NAAM            TYPE = CHARACTER 20.
        ADRES           TYPE = CHARACTER 30.
        GEBDATUM        TYPE = CHARACTER 12.
```

De eerste regel van bovengenoemde bestandsdefinitie geeft aan dat WERKNEMERS de naam van het bestand is. In het bestand komen alleen WERKNEMER records voor. Een opgeslagen WERKNEMER record heeft een lengte van 70 karakters. Het bestand zal in het algemeen even veel records bevatten als er werknemers zijn.

Het WERKNEMERS bestand kan bijvoorbeeld worden gebruikt als adressenbestand voor de verzending van circulaires met betrekking tot rechtspositie en dergelijke. Ook kan het bestand gebruikt worden om lijsten te produceren waaruit de leeftijdsopbouw van de werknemers blijkt. Een en ander vereist een aantal programma's die werken op hetzelfde bestand of, algemener gesteld, op dezelfde groep van bestanden.

Omdat een bestand (een deel van) het object systeem moet weerspiegelen en omdat er in het object systeem veranderingen optreden, zullen bestanden moeten worden bijgewerkt. Werknemers worden aangenomen, ontslagen, gaan verhuizen, etc. Dit resulteert in het bijwerken van het WERKNEMERS bestand. Onder bijwerken wordt verstaan het toevoegen (als nieuwe werknemers zijn aangenomen), verwijderen (als werknemers het bedrijf verlaten hebben) dan wel wijzigen (als bijvoorbeeld een verhuizing heeft plaatsgevonden) van opgeslagen records.

Op een bestand kunnen dus verschillende programma's werken. In Fig. 1.3 worden een aantal gebruikers getoond die via programma's op een aantal bestanden werken. (Fig. 1.3 is zeer schematisch; met name is het beheerssysteem dat de communicatie met de opslagmedia verzorgt, weggelaten.) Sommige programma's werken op één, andere op meer dan één bestand. De gebruikers zullen hun wensen moeten formuleren ten aanzien van de snelheid waarmee de diverse

programma's (bijvoorbeeld het leeftijdsopbouwprogramma en het adressenprogramma) uitgevoerd moeten worden.

Beschouw nu het probleem van de opslag van het WERKNEMERS bestand. Er zijn in de loop der jaren een aantal bestandsorganisaties ontwikkeld die het mogelijk maken om records van hetzelfde type op geschikte wijze op te slaan en te manipuleren. Iedere organisatie heeft zijn eigen karakteristieken ten aanzien van:

1. De snelheid waarmee leesoperaties uitgevoerd kunnen worden ("Geef de werknemer met nummer 234517").
2. De snelheid waarmee bijwerkoperaties worden uitgevoerd, dit weer uitgesplitst naar het type bijwerkoperatie (invoegen, verwijderen of wijzigen).
3. Het extern geheugengebruik: sommige bestandsorganisaties hebben extra tabellen nodig, andere niet.

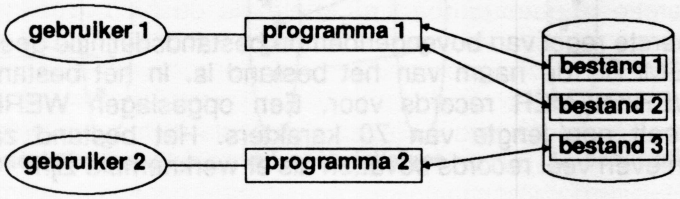



**Fig. 1.3 De bestandsaanpak.**

De keuze van de optimale bestandsorganisatie is een moeilijk probleem, dat buiten het kader van dit boek valt. Laten we aannemen dat gekozen wordt voor een geïndexeerd sequentiële bestandsorganisatie. (Hoewel we aannemen dat de meeste lezers vertrouwd zijn met bestandsorganisaties wordt er in hoofdstuk 11 kort op ingegaan.) De bestandsdefinitie zou er dan (schematisch) als volgt uit kunnen zien:

```
WERKNEMERS BESTAND
     ORGANISATIE IS GEINDEXEERD SEQUENTIEEL,
     SLEUTEL IS WERKNO.
WERKNEMER RECORD
          LENGTE = 70 CHARACTERS.
          WERKNO          TYPE = CHARACTER 8.
          NAAM            TYPE = CHARACTER 20.
          ADRES           TYPE = CHARACTER 30.
          GEBDATUM        TYPE = CHARACTER 12.
```

Genoemd bestand is opgeslagen volgens de geïndexeerd sequentiële bestandsorganisatie, waarbij de sleutel WERKNO is. De waarde van een *sleutel* identificeert een record; zo identificeert de waarde voor het veld WERKNO het WERKNEMER record waarin die waarde voorkomt. (Op het begrip sleutel wordt in dit boek nog uitvoerig teruggekomen.) De programma's die met dit bestand werken, zullen nu (programmeertaalafhankelijke) opdrachten gaan bevatten waarmee het bestand gemanipuleerd kan worden. Dat betekent dat in de programma's opdrachten met betrekking tot de geïndexeerd sequentiële bestandsorganisatie zullen voorkomen. Analoge opmerkingen gelden als niet de geïndexeerd sequentiële, maar een hashed of sequentiële bestandsorganisatie gekozen zou zijn. Belangrijk is dat, wat ook gekozen wordt, de *programma's* die op het *bestand werken, beïnvloed zijn door de keuze van de bestandsorganisatie*. Dit is een zeer belangrijk punt waarop we uitvoeriger zullen ingaan.

**1.3.1 Fysieke gegevensafhankelijkheid.** In de hierboven geschetste aanpak is de manier waarop de gegevens op secondair geheugen (bijvoorbeeld schijf) zijn georganiseerd en de manier waarop toegang tot deze gegevens (direct of sequentieel, zie hierna) wordt verkregen van invloed op de programma's die op het bestand moeten werken. Met andere woorden, *kennis van die organisatie en van de toegangstechniek moet in de programmalogica en -code verwerkt worden.* Het WERKNEMERS bestand is geïndexeerd sequentieel georganiseerd met WERKNO als sleutel. Leesoperaties met een WERKNO waarde als parameter kunnen dan snel beantwoord worden (directe toegang). Bovendien kunnen de records van het bestand gesorteerd op WERKNO worden gelezen (sequentiële toegang). Een programmeur die een programma moet schrijven dat op het WERKNEMERS bestand werkt, zal zich in het algemeen van deze twee zaken bewust zijn en de interne structuur van het programma rond deze kennis opbouwen. In het bijzonder zullen de procedures die de toegang tot het bestand regelen en de procedures die uitzonderingssituaties afhandelen ("einde van bestand", "record niet gevonden", en dergelijke), zeer sterk beïnvloed worden door de keuze van de bestandsorganisatie. We zeggen dat een dergelijk programma gegevensafhankelijk is, omdat het onmogelijk is de bestandsorganisatie of de toegangstechniek te veranderen zonder het programma te wijzigen. Het is bijvoorbeeld onmogelijk om het WERKNEMERS bestand van een geïndexeerd sequentieel bestand te wijzigen in een hashed bestand zonder belangrijke wijzigingen in het programma aan te brengen. Nog belangrijker is dat het deel van het programma dat gewijzigd moet worden niets met de vraagstelling van de gebruiker te maken heeft, maar alleen met de opslag van gegevens. Aangezien de besproken afhankelijkheid zich afspeelt op het opslagniveau ("dichtbij" de fysieke opslagmedia) wordt de term "fysieke gegevensafhankelijkheid" hiervoor gebruikt. Later zullen we ook een logische gegevensafhankelijkheid invoeren (zie paragraaf 2.6).

We gaan nu nader in op de soorten wijzigingen die zich in de opslagstructuren kunnen voordoen. Onderstaande lijst is bij lange na niet volledig, maar geeft toch een indruk van de afhankelijkheden die kunnen optreden.

*Op veld niveau:*

- Een karakter kan in EBCDIC, ASCII of een andere code zijn gerepresenteerd. Veranderingen in deze representatie veroorzaken veranderingen in de programmatuur, omdat in de programma's deze representatie is opgenomen.

- Een numeriek veld kan worden opgeslagen in een intern formaat of als een karakterstring. In beide gevallen zal een geschikt grondtal (bijvoorbeeld twee of zestien) en een geschikte schaal (vaste of drijvende komma), mode (reëel of complex getal) en precisie (aantal cijfers) moeten worden gekozen. Al deze aspecten zijn aan veranderingen onderhevig. Zodra een verandering optreedt zullen de programma's die van de oude representatie uitgaan moeten worden aangepast.

- De eenheden in een numeriek veld kunnen veranderen. Zo kunnen er conversies nodig zijn van yards naar meters, van guldens naar ECU (European Currency Unit), etc. Programma's die op gegevens uitgedrukt in de oude eenheden werken, zullen door deze conversies worden beïnvloed.

*Op record niveau:*

- Twee bestaande opgeslagen records kunnen worden samengevoegd tot één. Stel dat er twee bestanden bestaan waarin records zijn opgeslagen met respectievelijk als velden (WERKNO, ADRES) en (WERKNO, GEBDATUM). Dan zouden deze bestanden geïntegreerd kunnen worden tot één bestand met records waarin de drie velden tezamen zijn opgenomen.
- Het omgekeerde kan ook het geval zijn. Stel dat er een bestand is met records die bestaan uit drie velden, waarvan de eerste twee heel vaak worden gebruikt, maar het derde veld, daarentegen, nauwelijks. Als men de *prestatie* van het systeem nu wil verbeteren, dan zou men het bestand in tweeën kunnen splitsen en het veel gebruikte deel op een snel medium kunnen zetten zodat de verwerking ervan snel kan geschieden.

Ook hier geldt weer: de programma's die op deze bestanden werken, moeten worden aangepast als het recordtype wijzigt.

*Op bestand niveau:*
Een bestand kan op vele manieren opgeslagen worden.
- Geordend op een zeker veld of op een groep van velden. Een programma dat hier vanuit gaat, zal moeten worden aangepast als deze ordening verdwijnt.
- In een bestand kunnen ketens worden bijgehouden om bepaalde vragen snel te kunnen beantwoorden. Stel dat de afdeling Personeelszaken van een bedrijf vaak een lijst wil zien van werknemers met een bepaalde geboortedatum. Ketening van alle werknemers met dezelfde geboortedatum zou het produceren van een dergelijke lijst drastisch kunnen versnellen. Als een programma is geschreven met als aanname het bestaan van deze keten, dan zal het gewijzigd moeten worden zodra de keten verdwijnt.

Deze lijst van mogelijke datastructureringen is zeer lang te maken. Bestanden kunnen worden opgebouwd met behulp van adresberekeningstechnieken, indexen, etc. (zie ook hoofdstuk 11). Zodra, om welke reden dan ook, in de opslagstructuur een wijziging komt, zal dit gevolgen hebben voor de programma's die de bestanden gebruiken.

*Waarom is fysieke gegevensafhankelijkheid ongewenst?* Sinds enige decennia is er een tendens dat computerapparatuur goedkoper en (automatiserings)personeel duurder wordt. Het ligt daarom voor de hand om te proberen te besparen op personeel, ook al zou dat eventueel (extra) kosten voor apparatuur met zich meebrengen. Nu is het onderhouden van programma's een taak, die een groot beslag op het personeel legt. Programma-onderhoud wordt onder andere veroorzaakt doordat de programma's niet onafhankelijk zijn van de opslag van gegevens. Dat betekent dat naar een middel moet worden gezocht om van de diverse vormen van fysieke gegevensafhankelijkheid af te komen. Dit wordt nu verkregen door een extra programmatuurlaag aan te brengen, waardoor de gegevensafhankelijkheden worden weggewerkt. Dit nu zal een taak blijken te zijn van het database management systeem.

**1.3.2 Toepassingsgebonden bestanden.** Zoals reeds werd opgemerkt, worden bestanden in het algemeen ontworpen ten dienste van een beperkt aantal programma's die tezamen een bepaalde functie vervullen of, anders gezegd, een toepassing vormen. Deze bestanden noemen we dan ook "toepassingsgebonden". Bij verschillende toepassingen worden dikwijls (deels) dezelfde gegevens gebruikt. Hiervoor werd een toepassing beschreven die gegevens over werknemers gebruikte. Maar die gegevens zijn ook nodig bij de projectplanning. De inkoopadministratie werkt met gegevens over artikelen, maar een toepassing betreffende produktievoorbereiding doet dat ook, etc. De gegevens zijn steeds ondergebracht in bestanden die

'eigendom' zijn van de toepassing die ze gebruikt en beheert. Gegevens met betrekking tot een zekere entiteit (bijvoorbeeld een artikel) kunnen dus vele malen worden opgeslagen, soms zelfs tot tientallen malen toe. Dit verschijnsel noemt men *gegevensredundantie*.

*Wat zijn de nadelen van toepassingsgebonden bestanden?* Een reeds genoemd nadeel is dat gegevensredundantie optreedt, met als gevolg dat veel meer geheugenruimte gebruikt wordt dan strikt nodig is. Een ander zeer belangrijk nadeel is het gevaar van inconsistentie. Een gegevensverzameling is consistent (vrij van tegenspraak) als er niet twee (of meer) feiten uit afleidbaar zijn die met elkaar in strijd zijn. Bij gegevensredundantie wordt eenzelfde kenmerk van een entiteit meermalen opgeslagen. Hierdoor bestaat het gevaar dat voor dat kenmerk verschillende waarden worden opgeslagen. Zo kan in het ene bestand Enschede als adres voor een werknemer genoteerd staan, terwijl dat in een ander bestand Utrecht is. Dit is een vorm van inconsistentie, die in de hand wordt gewerkt door redundantie.

Een oplossing voor de geschetste problemen is het *centrale beheer* van alle gegevens die voor een bedrijf van belang zijn. Dit kan worden bevorderd door het instellen van de functie van een database administrator (afgekort DBA, in het Nederlands ook wel databasebeheerder genoemd). De uitoefening van deze functie (die door een of meer werknemers vervuld kan worden) houdt onder andere in het onderkennen van het gebruik van dezelfde gegevens door vele toepassingen en het nemen van adequate maatregelen dienaangaande.

## 1.4 WAT IS EEN DATABASE SYSTEEM?

Een database systeem is in essentie niets anders dan een systeem dat met behulp van een computer de bedrijfsgegevens beheert en op aanvraag ter beschikking stelt. Zo'n systeem bevat vier componenten, namelijk de gebruikers, de apparatuur, de gegevens zelf en het database management systeem, zie Fig. 1.4 (ook hier is het beheerssysteem weggelaten; zie ook Fig. 11.2).

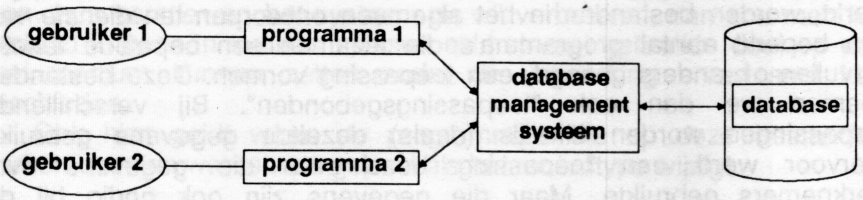



**Fig. 1.4 De databaseaanpak.**

De *gebruikers* worden onderscheiden in programmeurs die in programmeertalen als PASCAL en COBOL met een database werken, en zogenaamde eindgebruikers die via vraagtalen (zie hoofdstuk 7) de database bijwerken dan wel er gegevens aan onttrekken.

De *apparatuur* bestaat uit externe geheugenmedia (schijvengeheugens), waarop de database fysiek is opgeslagen tezamen met de geassocieerde schijveneenheden, controllers, kanalen, etc. We gaan hier niet nader op in, aangezien apparatuur buiten de scope van dit boek valt (wel wordt in paragraaf 11.1 een korte schets van schijvengeheugens gegeven).

De *gegevens* in de database zijn *geïntegreerd en gedeeld*. Deze twee kenmerken vormen het onderscheid tussen een database en een verzameling toepassingsgebonden bestanden zoals in de vorige paragraaf beschreven. Met geïntegreerd en gedeeld wordt het volgende bedoeld.

- *Geïntegreerd* betekent dat de database kan worden beschouwd als de vereniging van verscheidene bestanden, waarbij iedere redundantie tussen die bestanden geheel of gedeeltelijk is geëlimineerd. Een database zou bijvoorbeeld informatie over werknemers (naam, adres, geboortedatum, etc.) en inschrijvingen van die werknemers voor cursussen kunnen bevatten. Veronderstel dat de cursusadministratie over de adressen van de cursisten moet beschikken. Dan is er geen noodzaak om die informatie apart bij de inschrijvingen te noteren aangezien de informatie al bekend is, dat wil zeggen, aanwezig is in de informatie betreffende de werknemers. Werd er met afzonderlijke bestanden gewerkt, dan zou het adres zowel in een WERKNEMERS als een in INSCHRIJVINGEN bestand zijn opgenomen.

- Met *gedeeld* bedoelen we dat delen van de database door meerdere gebruikers mogen worden gebruikt, in die zin dat elk van die gebruikers toegang tot eenzelfde deel mag hebben (en dat deze gebruikers het deel voor verschillende doelen mogen gebruiken). Het is zelfs zo dat verschillende gebruikers *op hetzelfde moment* toegang mogen hebben tot eenzelfde deel van de database ("gelijktijdig gebruik"). Dit gedeelde gebruik (gelijktijdig of niet) is ten dele een consequentie van het geïntegreerde karakter van een database. In het hierboven genoemde voorbeeld zou het adres van werknemers gedeeld worden door gebruikers uit de afdeling Personeelszaken en gebruikers uit de afdeling Opleidingen; de ene keer om circulaires van adressen te voorzien en een andere keer voor de cursusadministratie.

Een consequentie van het feit dat de database geïntegreerd is, is dat een gebruiker normaal gesproken slechts in een deelverzameling van de database geïnteresseerd is. Een gegeven database zal door de diverse gebruikers verschillend worden bekeken. Hierop wordt later uitvoerig teruggekomen.

Het *database management systeem* (het DBMS) is de programmatuur tussen de database zelf en de gebruikers van de database. Alle aanvragen van gebruikers voor de database worden door het DBMS afgehandeld. Hierdoor worden onder andere lees- en bijwerkoperaties uitgevoerd. Een belangrijke functie van het DBMS is de gebruikers af te schermen van de details die te maken hebben met de apparatuur en met de opslag van gegevens. Om het anders te zeggen: het DBMS moet *fysieke gegevensonafhankelijkheid* aanbieden. Daardoor hebben wijzigingen in de opslagstructuur geen invloed meer op programma's. Dit is een van de belangrijke doelstellingen van database systemen.

Fig. 1.4 toont de plaats van een DBMS tijdens de executie van programma's. We zien dat een programma met het DBMS communiceert en dat dit DBMS het contact met de database verzorgt. De DBMS programmatuur kan zo de fysieke gegevensonafhankelijkheid aanbieden.

## 1.5 VOORDELEN VAN DATABASES

Uit het voorgaande blijkt dat er twee belangrijke aspecten zijn waaruit de voordelen van het gebruik van database systemen voortvloeien. Het eerste aspect is het bij een database systeem behorende centrale beheer van de bedrijfsgegevens, en het tweede aspect is het bieden van fysieke gegevensonafhankelijkheid.

**1.5.1 Centraal Beheer.** Het centrale beheer van de bedrijfsgegevens heeft vele gevolgen.

- De mate van redundantie kan drastisch worden verminderd.

In de bestandsaanpak heeft iedere toepassing zijn eigen bestanden, hetgeen dikwijls leidt tot een niet onaanzienlijke redundantie in de opgeslagen gegevens. Het resultaat is een verspilling van opslagruimte. In de bestandsaanpak zou het adres van werknemers, dat door een aantal toepassingen wordt gebruikt, even zovele malen opgeslagen zijn. Een database aanpak resulteert in het slechts eenmaal opslaan van het adres.

Voor alle duidelijkheid, het is niet de bedoeling dat alle redundantie koste wat het kost moet worden verwijderd. Er kunnen redenen zijn om

redundantie toe te laten, ook in een database benadering. Belangrijk is, dat de redundantie bekend moet zijn bij het DBMS, zodat eventueel adequate acties genomen kunnen worden. Het valt echter buiten de scope van dit boek om hier dieper op in te gaan (zie voor meer informatie bijvoorbeeld [BEHG87] en [DATE83]).

- Inconsistenties kunnen (tot op zekere hoogte) worden vermeden.

Dit is een gevolg van het vorige punt. Stel dat een zeker gegeven, bijvoorbeeld het gegeven dat werknemer W3 in afdeling A8 werkt, tweemaal in de database is opgeslagen. Veronderstel eveneens dat het DBMS dit niet weet. Dan zullen zich momenten voordoen dat deze duplicaten niet overeenstemmen, namelijk op die momenten dat slechts één van de twee is bijgewerkt. Op die momenten is de database inconsistent. Het is duidelijk dat deze situatie ongewenst is. Als een gegeven slechts eenmaal in de database is opgeslagen kan inconsistentie niet optreden.

Als een DBMS een gegeven toch redundant opslaat (bijvoorbeeld om prestatie-redenen), dan heeft het DBMS ook de verantwoordelijkheid om de gebruikers steeds een consistente database te tonen.

- Opgeslagen gegevens kunnen worden gedeeld.

Dit punt werd reeds besproken in de vorige paragraaf maar voor de volledigheid wordt het hier nog een keer genoemd. Deling betekent niet alleen dat bestaande toepassingen dezelfde gegevens in de database kunnen gebruiken, maar ook dat nieuwe toepassingen kunnen worden ontwikkeld die op dezelfde gegevens werken. Met andere woorden, het is (vaak) mogelijk om de gegevensbehoefte van nieuwe toepassingen te vervullen, zonder dat additionele gegevens behoeven te worden gecreëerd.

- Standaards kunnen worden afgedwongen.

De centrale beheerstaak wordt toegewezen aan de DBA. De DBA kan ervoor zorgen dat men zich bij het representeren van gegevens houdt aan alle van belang zijnde standaards. Deze standaards kunnen bijvoorbeeld worden opgelegd door het bedrijf, een afdeling, het rekencentrum of door nationale of internationale instellingen. Het standaardiseren van gegevensformaten is in het bijzonder van belang bij gegevensuitwisseling. Op dezelfde wijze zijn regels ten aanzien van naamgeving en documentatie van belang bij het gemeenschappelijk gebruik en het "begrijpen" van gegevens.

- Beveiligingsregels kunnen beter worden toegepast.

Omdat hij volledige zeggenschap over de gegevens heeft, kan de DBA:

a) ervoor zorgen dat men alleen via de daarvoor geëigende wegen toegang tot gegevens kan krijgen, en

b) beveiligingscontroles definiëren die worden uitgevoerd zodra men probeert toegang te krijgen tot zogenaamde gevoelige gegevens (bijvoorbeeld gegevens betreffende geloofsovertuiging, salaris, etc.).

Verschillende controles kunnen worden ingebouwd voor ieder type toegang (lezen, invoegen, wijzigen, verwijderen) tot ieder deel van de database.

- Integriteit kan beter worden gehandhaafd.

De integriteit van de database is de mate waarin de gegevens in de database accuraat zijn, dat wil zeggen de mate waarin ze overeenstemmen met de "feiten" in het object systeem waarvan zij een representatie moeten zijn. Inconsistentie als gevolg van het tweemaal opslaan van een gegeven is een voorbeeld van een gebrek aan integriteit. Natuurlijk kan dit probleem alleen optreden als er redundantie in de database aanwezig is. Zelfs als er geen redundantie is, kan de database incorrecte informatie bevatten. Zo kan, volgens de database, een werknemer in een week 400 in plaats van 40 uur gewerkt hebben of is hij aan een niet bestaande afdeling toegewezen. Centraal beheer van de database is een hulpmiddel bij het tegengaan van zulke problemen. Dit betekent in de praktijk dat de DBA zogenaamde integriteitscontroles kan definiëren. Deze controles worden door het DBMS uitgevoerd zodra een programma een bijwerkoperatie uitvoert. Een dergelijke controle zou kunnen inhouden dat het aantal te werken uren per week nooit meer dan 50 mag bedragen. Hiermee kunnen sommige fouten (bijvoorbeeld bovengenoemde fout) worden vermeden, andere echter niet. Stel dat iemand invoert dat een werknemer 20 uur heeft gewerkt, terwijl het juiste aantal 25 is. Dat wordt met genoemde controle natuurlijk niet ontdekt.

- Conflicterende eisen kunnen worden afgewogen.

Uitgaande van het feit dat hij alle eisen van het bedrijf kent, kan de DBA de database zo structureren dat de service aan de gebruikers in het bedrijf als geheel wordt geoptimaliseerd. Gegevens kunnen namelijk zo opgeslagen worden dat de belangrijkste toepassingen het best bediend worden, hetgeen dan wel ten koste kan gaan van de bediening die sommige andere toepassingen krijgen.

**1.5.2 Fysieke gegevensonafhankelijkheid.** Op gegevensafhankelijkheid is al uitvoerig ingegaan. Gegevensonafhankelijkheid kan worden

gedefinieerd als het "immuun" zijn van programma's voor veranderingen in opslagstructuur en toegangstechniek. Hieronder volgen een tweetal consequenties van deze immuniteit.

- Het probleemloos kunnen wijzigen van opslagstructuur en toegangstechnieken.

Hier wordt onder verstaan dat de DBA de vrijheid heeft om de opslagstructuur en/of de toegangstechnieken te wijzigen als antwoord op zich wijzigende omstandigheden zonder dat de bestaande toepassingen behoeven te worden aangepast. Deze zich wijzigende omstandigheden kunnen zijn: de komst van nieuwe (snellere) apparatuur, het belangrijk worden van voorheen als onbelangrijk beschouwde toepassingen, nieuwe gegevens die moeten worden opgenomen in de database en die toegangspatronen op externe media drastisch doen wijzigen, etc. Deze wijzigingen hebben in de database aanpak geen gevolgen voor de programmatuur, zodat programmeurs zich niet met deze aanpassingen hoeven bezig te houden.

- Het toestaan dat toepassingen eenzelfde gegeven als verschillend opgeslagen beschouwen.

Stel dat twee toepassingen A en B een veld SALARIS gebruiken en neem aan dat A deze waarde als decimaal en B als binair ziet (dat wil zeggen als zodanig heeft gedefinieerd). Dit kan in een database omgeving zeer goed samengaan, omdat het DBMS eventuele conversies voor zijn rekening neemt. Stel dat SALARIS binair is opgeslagen. Dan moet iedere keer als A dit veld wil lezen een conversie naar decimaal plaatsvinden en bij schrijven moet de omgekeerde conversie geschieden.

Dit is een eenvoudig voorbeeld van het verschil dat mag bestaan tussen de opslag van gegevens en de manier waarop toepassingen gegevens wensen te zien.

## 1.6 NADELEN VAN DATABASES

Vanzelfsprekend kent het gebruik van database systemen ook nadelen. De database benadering eist een centrale aanpak en het in het leven roepen van de DBA functie. Dit houdt verlies van lokale autonomie in. Bovendien kunnen starheid en te veel bureaucratie (kwalijke) gevolgen van de centrale aanpak zijn. Hierna worden (net als hiervoor weer vergeleken met de bestandsaanpak) enige nadelen van databases genoemd.

- Grotere afhankelijkheid van het goed functioneren van het computersysteem.

Bij de database aanpak is bedrijfszekerheid van nog groter belang dan bij de bestandsaanpak. Dit omdat een groter deel van de bedrijfsoperaties er van afhangt. Dit geldt minder voor de toepassingsgerichte aanpak, omdat het niet of slecht functioneren van een toepassing beperktere gevolgen voor het bedrijf heeft.

- Hinder van andere gebruikers.

Gebruikers kunnen elkaar op vele wijzen hinderen. We zullen slechts op een drietal vormen ingaan. Door de database aanpak werken vele gebruikers op dezelfde gegevens. Hierdoor zijn beveiligingsmaatregelen nodig aangezien niet iedere gebruiker vrijelijk toegang tot alle gegevens mag hebben. Natuurlijk zullen deze maatregelen niet altijd afdoende zijn, zodat ongeoorloofd en hinderlijk gebruik van gegevens door derden mogelijk blijft.

Iets dergelijks geldt ook ten aanzien van integriteit. Als in een database aanpak foutieve gegevens worden ingevoerd, dan zal dat gevolgen hebben voor andere, 'onschuldige' gebruikers van het systeem.

Een derde vorm van hinder kan optreden bij het gelijktijdig gebruik van een database. Er dreigt dan gevaar voor *deadlock* (dodelijke omarming, impasse). Een deadlock situatie houdt in dat de uitvoering van een aantal programma's geen voortgang kan hebben omdat ze op elkaar wachten. De betreffende programma's komen zonder hulp van het DBMS hier niet meer uit. Deze hulp houdt in dat een programma wordt gestopt, waarna een herstart van het programma plaats vindt. Dit wordt door de gebruiker van dat programma natuurlijk als vervelend ervaren.

- Prestatiedaling voor individuele gebruikers.

Bij de voordelen is gezegd dat de DBA de voor een bedrijf optimale opslagstructuur kan kiezen. Dit is juist. Als echter iedere toepassing zijn eigen bestanden heeft, dan kan de voor deze toepassing optimale opslagstructuur en toegangstechniek worden gekozen. Dit houdt in, dat in de database benadering de prestatie voor iedere toepassing afzonderlijk in het algemeen slechter zal zijn dan bij de bestandsaanpak.

- Kosten.

Aan het gebruik van database management systemen zijn kosten verbonden. De aanschaf van een DBMS kan, zeker op de grotere computersystemen, een behoorlijke uitgave betekenen. Daarnaast kan

het gebruik van database systemen nogal wat computer resources vragen (processortijd, intern geheugen, etc.). Bovendien zijn ze veel complexer dan bestanden, hetgeen inhoudt dat veel kennis opgebouwd moet worden om databases adequaat te kunnen onderhouden en gebruiken. Dit betekent dat nieuw (en vaak duur) personeel moet worden aangetrokken, dan wel dat aanwezig personeel bijgeschoold moet worden.

## OPGAVEN

De inhoud van dit hoofdstuk bevat vele begrippen. Om te testen of u die begrippen hebt begrepen, wordt u verzocht de volgende vragen te beantwoorden.

**1.1** Definieer de volgende termen.

object systeem
entiteit en entiteittype
associatie en associatietype
consistentie, integriteit en beveiliging
database administrator (DBA)
database management systeem

**1.2** Geef een voorbeeld van een associatietype dat een verband tussen twee entiteittypen definieert en waarbij niet alle entiteiten van die twee typen in een associatie van genoemd type betrokken zijn.

**1.3** Beschrijf de voor- en nadelen van databases.

# 2
# De Architectuur van een Database Systeem

## 2.1 INLEIDING

De gegevens in een database worden uiteindelijk in computergeheugens opgeslagen in de vorm van rijen bits. Het is duidelijk dat er tussen deze bits en de uiteindelijke gebruiker, die denkt in termen van leveranciers en artikelen, een aantal niveaus van abstractie zijn. Een architectuur van een database systeem geeft aan welke niveau's er zoal zijn en hoe de afbeeldingen tussen die niveau's er uit zien. In dit hoofdstuk wordt een heden ten dage als zinvol beschouwde architectuur beschreven, namelijk een architectuur die in grote lijnen overeenkomt met het voorstel van de ANSI/SPARC Study Group on Data Base Management Systems (zie ook [ANSI75] en [TSKL78]). Het doel van deze beschrijving is om een raamwerk te bieden dat in de komende hoofdstukken gebruikt kan worden bij het invoeren van database concepten.

## 2.2 DE DRIE NIVEAUS VAN DE ARCHITECTUUR

De architectuur is verdeeld in drie niveaus: het interne, het conceptuele en het externe niveau (zie Fig. 2.1). Globaal gesproken:

1. ligt het *interne* niveau het dichtst bij de fysieke opslag, dat wil zeggen, heeft van doen met de manier waarop de gegevens opgeslagen zijn;

2. staat het *externe* niveau het dichtst bij de gebruikers, dat wil zeggen, heeft te maken met de wijze waarop de gegevens door de individuele gebruikers worden gezien; en
3. is het *conceptuele* niveau een 'niveau van indirectie' tussen de andere twee.

Het externe niveau houdt zich bezig met de visie van de *individuele* gebruikers, terwijl het conceptuele niveau gezien moet worden als het niveau dat te maken heeft met de *gemeenschappelijke* visie van alle gebruikers. Met andere woorden, er zullen in het algemeen *vele 'externe visies'* zijn, elk bestaande uit een min of meer abstracte representatie van een deel van de totale database. Anderzijds zal er precies *één 'conceptuele visie'* zijn, bestaande uit een even abstracte representatie van de database als geheel. (Hierbij zij opgemerkt dat de meeste gebruikers niet geïnteresseerd zijn in de database als geheel, doch slechts in een beperkt deel daarvan.) Evenzo zal er slechts *één 'interne visie'* zijn, die de totale database zoals hij is opgeslagen, weergeeft. Merk op dat deze interne visie in het algemeen een nog vrij abstracte beschrijving is van de bits waar hierboven over gesproken is.

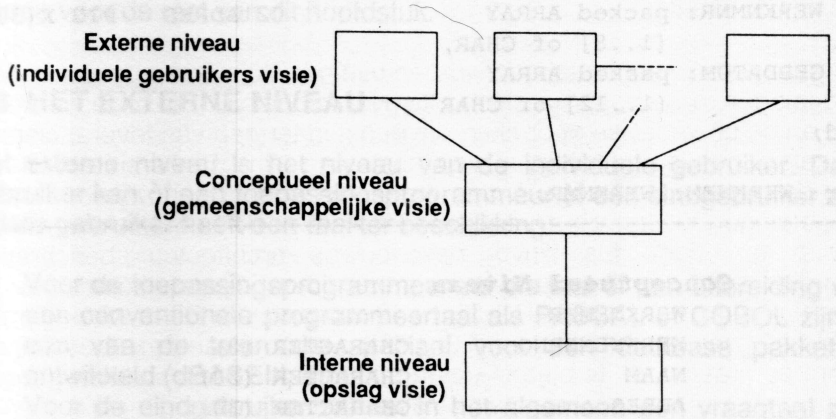



**Fig. 2.1 De drie niveaus van de architectuur.**

Een voorbeeld zal een en ander verduidelijken. Fig. 2.2 toont een personeelsdatabase zoals gezien op de drie niveaus. Het voorbeeld is zeer eenvoudig, enigszins hypothetisch en vele details zijn weggelaten. Op extern niveau zien we twee "gebruikers", namelijk een PASCAL en een COBOL gebruiker.

We interpreteren Fig. 2.2 als volgt.

- Op het conceptuele niveau bevat de database informatie over een entiteittype WERKNEMER. Iedere WERKNEMER heeft een

WERKNEMERNO (acht karakters), een NAAM (20 karakters), een ADRES (30 karakters) en een GEBDATUM (12 karakters).

- Op het interne niveau worden werknemers gerepresenteerd door een opgeslagen record genaamd OPGESLAGEN_WERKN dat 76 karakters lang is. OPGESLAGEN_WERKN bevat vijf opgeslagen velden: een zes karakters lang prefix (waarin wijzers en andere besturingsinformatie opgeslagen is) en vier velden die corresponderen met de vier kenmerken van de werknemers. Daarnaast is er een index WERKX gedefinieerd op het veld WERKNNUM.
- Voor de PASCAL gebruiker wordt iedere werknemer gerepresenteerd door een PASCAL record met twee velden (deze gebruiker is niet geïnteresseerd in de naam en het adres van werknemers). Het recordtype wordt gedefinieerd volgens de normale PASCAL regels.

***Extern Niveau*** *(PASCAL)*

```
type WERKNMR =

record
   WERKNMNR: packed ARRAY
             [1..8] of CHAR,
   GEBDATUM: packed ARRAY
             [1..12] of CHAR
end;

var  WERKNEM: WERKNMR;
```

***Extern Niveau*** *(COBOL)*

```
01 WERK_REC

  02 WERKNNO PIC X(8).
  02 ADRES   PIC X(30).
```

***Conceptueel Niveau***

```
WERKNEMER
   WERKNEMERNO      CHARACTER (8)
   NAAM             CHARACTER (20)
   ADRES            CHARACTER (30)
   GEBDATUM         CHARACTER (12)
```

***Intern Niveau***

```
OPGESLAGEN_WERKN  LENGTE = 76
PREFIX            TYPE=BYTE(6), OFFSET=0,
WERKNNUM          TYPE=BYTE(8), OFFSET=6,
                  INDEX=WERKX
NAAM              TYPE=BYTE(20), OFFSET=14,
ADRES             TYPE=BYTE(30), OFFSET=34,
GEBDATUM          TYPE=BYTE(12), OFFSET=64
```

**Fig. 2.2 Een voorbeeld van de drie niveaus.**

- Op analoge wijze wordt voor de COBOL gebruiker een werknemer voorgesteld door een COBOL record met, opnieuw, twee velden (dit keer zijn naam en leeftijd niet van belang). Het recordtype is gedefinieerd in overeenstemming met de normale COBOL regels.
- Merk op dat gelijke of overeenstemmende objecten verschillende namen kunnen hebben in de diverse schema's. Zo wordt bijvoorbeeld het werknemernummer aangeduid met WERKNMNR door de PASCAL gebruiker, WERKNNO door de COBOL gebruiker en WERKNEMERNO en WERKNNUM op conceptueel respectievelijk intern niveau. Het database systeem moet zich van deze correspondenties natuurlijk bewust zijn. Zo moet vastgelegd worden dat het COBOL WERKNNO veld overeenkomt met het conceptuele veld WERKNEMERNO, dat op zijn beurt op intern niveau wordt voorgesteld door het opgeslagen veld WERKNNUM. Zulke correspondenties (of afbeeldingen) zijn niet in Fig. 2.2 opgenomen. Hierop komen we in dit hoofdstuk nog terug.

We gaan nu de drie niveaus van de architectuur meer in detail bekijken. Fig. 2.3 geeft de belangrijkste componenten van de architectuur met hun samenhang. Deze figuur zal als referentiekader dienen voor de rest van dit hoofdstuk.

## 2.3 HET EXTERNE NIVEAU

Het externe niveau is het niveau van de individuele gebruiker. Deze gebruiker kan òf een toepassingsprogrammeur òf een eindgebruiker zijn. Iedere gebruiker heeft een *taal* ter beschikking:

- Voor de toepassingsprogrammeur zal die taal òf een uitbreiding van een conventionele programmeertaal als PASCAL of COBOL zijn òf een van de talen die speciaal voor een database pakket is ontwikkeld (dBASEIII, etc).
- Voor de eindgebruiker zal het in het algemeen een vraagtaal zijn, bijvoorbeeld SQL (zie hoofdstuk 7).

Belangrijk is dat al deze talen een zogenaamde *data subtaal* (of data sublanguage, DSL, een ANSI/SPARC term, zie Fig. 2.3) bevatten, dat wil zeggen, een deel van de programmeertaal waarmee database objecten en operaties kunnen worden gedefinieerd. Met zegt dat de data subtaal ingebed is in de *hosttaal* (in het Nederlands soms de gastheertaal genoemd).

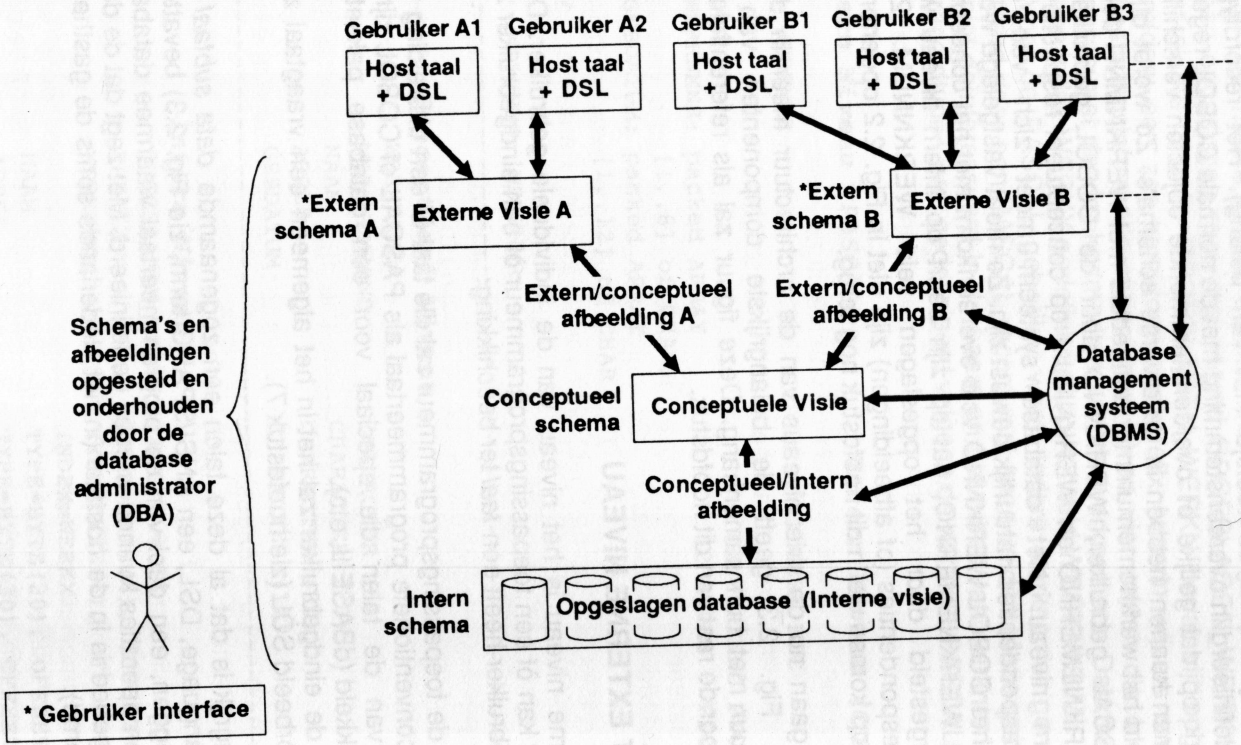



Fig. 2.3 Gedetailleerde systeemarchitectuur.

Deze hosttaal (denk bijvoorbeeld aan PASCAL of COBOL) biedt de niet-database faciliteiten aan, zoals lokale variabelen, rekenkundige operaties, if-then-else constructies, en dergelijke. Een gegeven database systeem kan vele hosttalen en data subtalen ondersteunen.

Beschouw de PASCAL gebruiker van Fig. 2.2. De data subtaal (data sublanguage, of DSL) voor deze gebruiker bestaat uit die PASCAL begrippen die worden gebruikt om met het DBMS te communiceren. Iedere DSL is een combinatie van een data definitietaal (data definition language, of DDL) en een data manipulatietaal (data manipulation language, of DML). Het DDL deel bestaat uit die PASCAL declaraties die nodig zijn om de database objecten te declareren: de type definitie en de declaratie van de variabele WERKNEM. Het DML deel bestaat uit die uitvoerbare PASCAL opdrachten die gegevens van en naar de database transporteren.

Om terug te keren naar de architectuur: we hebben reeds aangegeven dat een individuele gebruiker in het algemeen slechts in een deel van de totale database geïnteresseerd is; bovendien ziet de gebruiker dit deel niet in opgeslagen vorm, maar in een abstractie ervan. Voor de visie van een individuele gebruiker is de ANSI/SPARC term, zoals reeds eerder gezegd, een *externe visie*. Een externe visie is de inhoud van de database zoals die wordt gezien door een specifieke gebruiker (dat wil zeggen dat voor die gebruiker de externe visie *de database zelf* is). Zo kan een gebruiker van de afdeling Personeelszaken de database zien als een verzameling afdelingrecords plus een verzameling werknemerrecords en hij kan totaal onbewust zijn van het feit dat er ook nog leverancier- en artikelrecords zijn (die juist weer belangrijk zijn voor de gebruikers van de afdeling Inkoop).

In het algemeen bestaat een externe visie dus uit vele exemplaren van vele *externe recordtypen*. Een extern record is niet noodzakelijk hetzelfde als een opgeslagen record: er kunnen velden ontbreken, velden kunnen een ander type hebben (zie Fig. 2.2), etc. De DSL waarmee een gebruiker werkt, wordt gedefinieerd in termen van externe records; zo zal bijvoorbeeld een DML "lees record" operatie een extern record lezen en niet een opgeslagen record.

Iedere externe visie wordt gedefinieerd door middel van een *extern schema*, dat in essentie bestaat uit de definities van alle recordtypen die in de externe visie voorkomen. Het externe schema wordt geschreven in de DDL van de DSL van de gebruiker. In Fig. 2.2 zien we dat een extern record van type werknemer gedefinieerd wordt als een werknemernummer veld van acht karakters gevolgd door een geboortedatum veld van twaalf karakters. Belangrijk is dat de afbeelding tussen het externe schema en het onderliggende conceptuele schema moet worden gedefinieerd. Het conceptuele schema komt in de volgende paragraaf aan de orde, terwijl de afbeelding in paragraaf 2.6 ter sprake komt.

## 2.4 HET CONCEPTUELE NIVEAU

De *conceptuele visie* is de totale informatie-inhoud van de database, gerepresenteerd op een manier die geschikt is voor de gebruikersgemeenschap. Dit zal in het algemeen een andere manier zijn dan de manier waarop de gegevens opgeslagen zijn. Het kan ook verschillend zijn van de manier waarop een speciale gebruiker de database wenst te beschouwen. Een specifieke gebruiker heeft tot taak een bepaald probleem met behulp van de computer op te lossen. Dit probleem kan ten aanzien van data structurering speciale eigenschappen hebben; de structuur kan bijvoorbeeld hiërarchisch van aard zijn. Hierdoor en door de programmeerhulpmiddelen die gebruikt worden, kan de externe visie van een gebruiker worden beïnvloed. De conceptuele visie daarentegen heeft (globaal gesproken) ten doel de database te zien "zoals hij echt is", waarbij de zojuist geschetste invloeden niet van belang mogen zijn.

De conceptuele visie bestaat uit vele conceptuele records van verschillend type. De visie mag bijvoorbeeld bestaan uit afdelingrecords, werknemerrecords, leverancierrecords, artikelrecords, etc. Een conceptueel record is in het algemeen niet hetzelfde als een extern record of een opgeslagen record.

Wat de *conceptuele visie* is op *exemplaarniveau* is het *conceptuele schema* op *typeniveau*. Het conceptuele schema omvat typedefinities van alle records die in de database voorkomen. Het conceptuele schema wordt beschreven in een daarvoor geschikte DDL, de conceptuele DDL. Deze conceptuele DDL wordt gekozen op grond van het vermogen om het object systeem zo goed mogelijk te modelleren, wat inhoudt dat een dergelijke DDL rijk moet zijn aan semantische begrippen. Als gegevensonafhankelijkheid moet worden bereikt (en dat is natuurlijk een eis), dan mag het conceptuele schema geen zaken betreffende opslagstructuur en toegangstechnieken bevatten; er moet alleen aandacht geschonken worden aan de definitie van de informatie-inhoud. Daarom mag er geen verwijzing zijn naar de wijze waarop velden worden opgeslagen of naar het sorteren, indexeren of hashen van records. Als het conceptuele schema echt gegevensonafhankelijk is, dan zullen de externe schema's, die uitgedrukt zijn in termen van het conceptuele schema, ook gegevensonafhankelijk zijn.

De conceptuele visie is een visie van de hele database, en het conceptuele schema is een definitie van die visie. Het zou echter misleidend zijn om te beweren, dat het conceptuele schema niets anders is dan een verzameling recordbeschrijvingen zoals we die tegenkomen in bijvoorbeeld een COBOL programma. Het is de bedoeling dat de definities van het conceptuele schema additionele voorzieningen bevatten, met name ten behoeve van beveiligings- en integriteitscontroles. Er zijn zelfs database experts die suggereren dat het uiteindelijke doel van het conceptuele schema is om de volledige onderneming te beschrijven, niet alleen de bedrijfsgegevens op zich, maar ook hoe die gegevens worden gebruikt, hoe ze van het ene punt in

de onderneming naar het andere gaan, etc. Huidige database systemen ondersteunen deze laatste voorzieningen echter nog lang niet, al is een trend in deze richting aanwezig.

## 2.5 HET INTERNE NIVEAU

Het derde niveau van de architectuur is het interne niveau. De *interne visie* is de representatie van de hele database op laag niveau, dat wil zeggen, zij is gericht op de bits en bytes zoals die op de opslagmedia (schijven, etc) voorkomen. De interne visie bestaat uit vele, *interne records* van verschillend type. Hierbij is "intern record" de ANSI/SPARC term voor wat in hoofdstuk 1 steeds een opgeslagen record werd genoemd. Het interne niveau is dus nog een slag verwijderd van het apparatuurniveau, omdat we nog niet spreken over blokken op schijf, cilinders, sporen en dergelijke, zaken die op het allerlaagste niveau wel aan de orde komen. In principe neemt de interne visie aan dat er een oneindige lineaire adresruimte is. (Hoe deze adresruimte op de media wordt afgebeeld is erg systeemafhankelijk en wordt met opzet weggelaten uit de architectuur.)

Het interne niveau wordt beschreven met behulp van het interne schema, dat niet alleen de diverse opgeslagen recordtypen aangeeft, maar ook welke indexen er bestaan, hoe de opgeslagen records worden gerepresenteerd, in welke sorteervolgorde de records zijn opgeslagen (als er een is), etc. Het interne schema wordt beschreven in weer een andere taal, de interne DDL genaamd.

## 2.6 AFBEELDINGEN

In Fig. 2.3 komen twee soorten afbeeldingen voor: één soort heeft betrekking op de afbeelding tussen een extern niveau en het conceptuele niveau, de andere beeldt het conceptuele niveau af op het interne niveau. De *conceptuele/interne afbeelding* definieert de verbinding tussen de conceptuele visie en de opgeslagen database (=interne visie). De afbeelding geeft aan hoe conceptuele records en velden worden gerepresenteerd op het interne niveau. Als de structuur van de opgeslagen database wordt veranderd, dat wil zeggen als een verandering in de definitie van de opslagstructuur optreedt, dan zal de de conceptuele/interne afbeelding aangepast moeten worden. Het conceptuele schema blijft dan echter gelijk: daar is immers geen wijziging in opgetreden. (Het is de verantwoordelijkheid van de DBA om zulke veranderingen in de opslagstructuur te regelen.) Met andere woorden, de effecten van zulke wijzigingen blijven beperkt tot onder het conceptuele niveau. Op deze wijze wordt bereikt dat de programmatuur, die zich op het 'nog hogere' externe niveau bevindt, niet aangepast hoeft te worden. Zo wordt *fysieke gegevensonafhankelijkheid* bereikt.

Een *externe/conceptuele afbeelding* definieert de overeenkomst tussen een bepaalde externe visie en de conceptuele visie. De verschillen die kunnen bestaan tussen deze twee niveaus lijken op de verschillen tussen het conceptuele en het interne niveau. Zo kunnen velden op de twee niveaus van een verschillend datatype zijn, kunnen veld- en recordnamen verschillen, etc. Op een zeker moment kan een willekeurig aantal externe visies aanwezig zijn en een willekeurig aantal gebruikers dezelfde externe visie gebruiken (een externe visie is dus niet exclusief voor een gebruiker!). Externe visies mogen records of delen van records gemeenschappelijk hebben.

Een belangrijk punt is dat *uitbreiding* van het conceptuele schema (er komen bijvoorbeeld record- en/of veldtypen bij) wel invloed kan hebben op de afbeelding extern/conceptueel, maar geen invloed op de programmatuur die zich op het externe niveau bevindt. De aanpassing van de afbeelding is een zaak van de DBA. Het feit dat de programmatuur ongewijzigd kan blijven wordt *logische gegevensonafhankelijkheid* (of groei-onafhankelijkheid) genoemd. Deze vorm van gegevensonafhankelijkheid is zeer gewenst, omdat de *database door vele gebruikers wordt gedeeld.* In de klassieke situatie heeft iedere toepassing zijn eigen bestanden; wijzigingen die niets met deze bestanden te maken hebben, dringen vanzelfsprekend niet in de toepassingsprogramma's door.

## 2.7 SLOTOPMERKING

Naast middelen om gegevens te structureren zal een DDL in het algemeen ook faciliteiten bevatten om de database (zoveel mogelijk) *integer* te houden. Onder integriteit wordt verstaan de mate waarin de informatie die in de database is opgeslagen, overeenkomt met de werkelijkheid. Als in de database genoteerd staat dat werknemer Jansen in het jaar 1860 geboren is en als dat niet zo is, dan is de database niet integer. De DDL kan faciliteiten bieden om niet integere databasetoestanden (zoveel mogelijk) te vermijden. Met een dergelijke faciliteit zou bijvoorbeeld gespecificeerd kunnen worden dat een geboortejaar kleiner dan 1900 niet in de database geaccepteerd moet worden. Als echter zou worden ingevoerd dat Jansen in 1936 is geboren, terwijl dat 1938 zou moeten zijn, dan zal genoemde integriteitsregel natuurlijk geen soelaas bieden. Dus volledige integriteit wordt dan niet afgedwongen. Het deel van de DDL dat op integriteit betrekking heeft noemt men het *integriteitsdeel.* In onder andere de hoofdstukken 4 en 8 wordt hierop nader ingegaan.

De combinatie DDL en DML, die hierboven als de DSL van een hosttaal werd ingevoerd, wordt ook wel een *data model* (soms database model) genoemd. In dit boek worden twee data modellen besproken die voor het beschrijven van het conceptuele schema in aanmerking komen,

namelijk het relationele en het netwerk data model. Van beide data modellen kan men zich afvragen of ze voldoende uitdrukkingskracht, voldoende begrippen bevatten om het object systeem op adequate wijze te kunnen modelleren. Deze vraag zal door sommigen negatief beantwoord worden. Desondanks zullen we in dit boek het relationele model en het netwerk model als kandidaten voor het conceptuele niveau blijven beschouwen en wel om de volgende reden.

Stel dat we voor het modelleren van het object systeem een semantisch rijker data model wensen te gebruiken en dat we naar een database implementatie toe willen. Dan zou de volgende weg gevolgd kunnen worden. (Merk op dat er voor de meeste semantisch rijkere data modellen nog geen 'pakketten' bestaan.) Uitgaande van het object systeem als beschreven in dat semantisch rijkere data model, vindt een (vaak handmatige) vertaling plaats naar een beschrijving in het relationele of netwerk data model. Zodra we deze beschrijving hebben, is de weg naar een database implementatie open. De beschrijving in het relationele of netwerk data model wordt ook wel eens de *conceptuele operationele beschrijving* genoemd. Als in dit boek wordt gesproken over het conceptuele niveau, dan zal steeds dit conceptuele operationele niveau bedoeld worden, omdat in dit boek het accent ligt op het definiëren en gebruiken van databases en *niet* op het modelleren van het object systeem.

## OPGAVEN

**2.1** De ANSI/SPARC architectuur kent drie niveau's. Op welk niveau komen bestandsorganisaties aan de orde? En op welk niveau zal de definitie van uitvoerlijsten plaatsvinden?

**2.2** Geef aan welke stappen er gedaan moeten worden voordat een extern record ingelezen is. (Hoofdstuk 11 kan hierbij behulpzaam zijn.)

**2.3** Wat houden de begrippen logische en fysieke gegevensonafhankelijkheid in?

# 3
# Data Modellen: een Inleiding

## 3.1 INLEIDING

In het vorige hoofdstuk is het begrip data model ingevoerd. We kunnen database systemen indelen aan de hand van het gekozen data model. In dit hoofdstuk zal een drietal data modellen worden geïntroduceerd:

- het relationele data model,
- het hiërarchische data model, en
- het netwerk data model.

In het vorige hoofdstuk werd eveneens gesteld, dat sommige data modellen geschikter zijn dan andere om op een bepaald niveau van de ANSI/SPARC architectuur te functioneren. Genoemde drie data modellen zijn zonder meer geschikt om op het externe niveau dienst te doen. Het zal in dit hoofdstuk duidelijk worden dat bijvoorbeeld een gebruiker van het hiërarchisch data model anders tegen de opgeslagen gegevens aankijkt dan iemand die het relationele model gebruikt. Het netwerk en het relationele model zijn geschikt voor het conceptuele niveau; ook hierop zal in dit boek worden ingegaan. Hierbij zij nog eens verwezen naar paragraaf 2.7.

In dit hoofdstuk worden de data modellen zeer informeel ingevoerd. Een gevolg daarvan is wel dat de operatoren alleen in een soort pseudo-code gegeven kunnen worden. In latere hoofdstukken worden de manipulatietalen van het netwerk en het relationele data model uitvoerig behandeld.

## 3.2 HET RELATIONELE DATA MODEL

Deze en de volgende twee paragrafen maken gebruik van een voorbeeld database die betrekking heeft op leveranciers, artikelen en leverbare artikelen. Figuur 3.1 toont de inhoud van de database volgens de relationele visie.

De gegevens worden in drie tabellen gepresenteerd: L (leveranciers), A (artikelen) en LA (leverbare artikelen). De L tabel bevat voor iedere leverancier een leveranciernummer, naam, status code en woonplaats; de A tabel bevat voor elk artikel een artikelnummer, naam, kleur, gewicht en de plaats waar het artikel is opgeslagen; de LA tabel bevat voor iedere leverancier de artikelen uit zijn artikelenpakket (of anders gezegd, voor ieder artikel de leveranciers die het kunnen leveren). Dus een rij van de LA tabel bevat een leveranciernummer, een artikelnummer en de hoeveelheid waarin een levering zal plaatsvinden (bijvoorbeeld, als potjes jam door leverancier L1 in dozen van 200 stuks worden geleverd, dan is 200 de hoeveelheid). We formuleren dit ook wel als: leverancier LNO levert artikel ANO in de hoeveelheid HOEV.

We nemen aan dat iedere leverancier een uniek leveranciernummer heeft en precies één naam, status en woonplaats. Evenzo nemen we aan dat een artikel een uniek artikelnummer heeft en precies één naam, kleur, gewicht en plaats. Tenslotte: op elk moment komt het feit dat een leverancier een zeker artikel kan leveren slechts eenmaal voor.

L

| LNO | LNAAM | STATUS | WNPL |
|---|---|---|---|
| L1 | Smit | 20 | Londen |
| L2 | Janse | 10 | Parijs |
| L3 | Blaak | 30 | Parijs |

A

| ANO | ANAAM | KLEUR | GEW | PLAATS |
|---|---|---|---|---|
| A1 | Moer | Rood | 12 | Londen |
| A2 | Bout | Groen | 17 | Parijs |
| A3 | Schroef | Blauw | 17 | Rome |
| A4 | Schroef | Rood | 14 | Londen |

LA

| LNO | ANO | HOEV. |
|---|---|---|
| L1 | A1 | 300 |
| L1 | A2 | 200 |
| L1 | A3 | 400 |
| L2 | A1 | 300 |
| L2 | A2 | 400 |
| L3 | A2 | 200 |

**Fig. 3.1 De relationele visie.**

Elk van de drie tabellen lijkt sterk op een conventioneel sequentieel bestand, waarbij de rijen van de tabel overeenkomen met de records van het bestand en de kolommen met de velden van de records. Tabel A bijvoorbeeld bevat vier rijen (of records), ieder bestaand uit vijf kolommen (velden). Elk van deze tabellen is in feite een *representatie* van wat in de wiskunde bekend staat onder de naam *relatie*. Rijen van zulke tabellen worden in het relationele model gezien als *tupels* en kolommen worden met *attributen* aangeduid. De term relatie heeft een nauwkeurigere definitie dan de termen bestand of tabel. Er zijn echter belangrijke verschillen (zie voor een definitie van relatie hoofdstuk 4)

tussen relaties en tabellen of traditionele sequentiële bestanden. De relationele visie van gegevens, dat wil zeggen de visie waarbij het relationele data model wordt gebruikt, is gebaseerd op de observatie dat bestanden die aan zekere restricties voldoen, beschouwd mogen worden als relaties. Indien aan deze restricties wordt voldaan, mag de elementaire, relationele theorie worden toegepast op verscheidene praktische problemen die betrekking hebben op gegevens in zulke bestanden.

Attributen worden aangeduid door hun naam, voorafgegaan door de naam van de relatie. Bijvoorbeeld, L.LNO is de aanduiding van het attribuut LNO in L. Als er geen kans op misverstand is, dan mag de relatienaam worden weggelaten.

Er is echter een begrip in de relationele theorie, waarvoor geen analogon in de traditionele gegevensverwerking bestaat, namelijk het begrip *domein* (dit begrip is in Fig. 3.1 niet getekend). Een domein is een verzameling van waarden waaruit de actuele waarden moeten komen die een attribuut mag aannemen. Bijvoorbeeld, de waarden die het ANO attribuut (vanaf nu spreken we van attribuut in plaats van kolom) van de LA en de A relatie mag aannemen, moeten voorkomen in het domein van alle artikelnummers. Dit domein zal in het algemeen niet expliciet als een verzameling waarden in de database zijn opgeslagen; wel zal in het relevante schema het domein een naam hebben en zal de bijbehorende waardenverzameling zijn gedefinieerd (zie hoofdstuk 4). Attributen die op een domein gebaseerd zijn mogen dezelfde naam hebben als het domein, maar dat hoeft niet. (Het is duidelijk dat ze een verschillende naam moeten hebben als er dubbelzinnigheid zou optreden. Dit is bijvoorbeeld het geval als twee attributen in dezelfde relatie hun waarden uit hetzelfde domein putten. Hiervan wordt in hoofdstuk 4 een voorbeeld gegeven.)

Vanaf nu wordt aangenomen dat de attributen L.LNO en LA.LNO een gemeenschappelijk domein (namelijk de leveranciernummers) hebben en dat dit ook geldt voor A.ANO en LA.ANO (de artikelnummers). L.WNPL en A.PLAATS hebben hun waarden in het domein 'PLAATS' (dat is de verzameling van alle plaatsnamen). Nu volgt een zeer belangrijke karakteristiek van het relationele data model:

*Verbanden tussen tupels worden uitsluitend gerepresenteerd door attribuutwaarden van die tupels en deze waarden moeten komen uit een zelfde domein.*

Het feit dat leverancier L3 en artikel A2 in dezelfde plaats gelokaliseerd zijn, wordt bijvoorbeeld gerepresenteerd door het verschijnen van dezelfde waarde in het WNPL respectievelijk het PLAATS attribuut voor de betreffende twee tupels. Het is een karakteristiek van het relationele data model dat in de database zowel de informatie betreffende entiteiten alsook de informatie betreffende

associaties op een uniforme wijze is weergegeven, namelijk in de vorm van relaties. We zullen later zien, dat deze karakteristiek niet geldt voor het hiërarchische en netwerk data model.

We zullen niet nader op andere aspecten van de relationele structuur in gaan, dat gebeurt in hoofdstuk 4. Het moge echter duidelijk zijn dat de relationele structuur zeer eenvoudig te begrijpen is. Daarmee zijn we er nog niet. Voor de gebruiker is een relationele manipulatietaal, dus de verzameling van operatoren die wordt aangeboden als de gegevens in relaties worden gerepresenteerd, minstens zo belangrijk. We merken op dat de uniformiteit van de gegevensrepresentatie tot een overeenkomstige uniformiteit in de operatorenverzameling leidt: omdat informatie op slechts één wijze wordt voorgesteld hebben we slechts één operator nodig voor iedere bijwerkoperatie (invoegen, verwijderen, wijzigen). Dit is niet zo als informatie op verscheidene manieren kan worden gerepresenteerd. Zo zullen we in hoofdstuk 10 zien, dat het op het netwerkmodel gebaseerde DBTG systeem twee "invoegoperatoren" kent: STORE om een record in te voegen en CONNECT om een verbinding tussen twee records te bewerkstelligen.

Laten we nu enige specifieke operatoren bekijken. De "lees volgende met" operator leest het volgende tupel van een relatie dat voldoet aan een conditie die achter "met" is gespecificeerd. "Volgende" wordt geïnterpreteerd ten opzichte van de *huidige positie* (dat is normaal gesproken het tupel dat het laatst benaderd is; als startpositie wordt de positie voor het eerste tupel aangenomen). Deze operator wordt toegelicht in de voorbeelden van Fig. 3.2 waar de formulering van twee vragen is geschetst die betrekking hebben op de database van Fig. 3.1. De twee vragen zijn met opzet symmetrisch gekozen.

| V1 | V2 |
|---|---|
| Geef nummers van leveranciers die A2 kunnen leveren | Geef nummers van artikelen leverbaar door L2 |
| **while** nog een LA tupel **do**<br>lees volgende tupel<br>met ANO='A2';<br>druk LNO af;<br>**end**; | **while** nog een LA tupel **do**<br>lees volgende tupel<br>met LNO='L2';<br>druk ANO af;<br>**end**; |

**Fig. 3.2 Twee vragen voor de relationele visie.**

Voor wat de andere operatoren betreft, beperken we ons tot het beschouwen van drie eenvoudige problemen, namelijk het invoegen, verwijderen en wijzigen van een tupel. In elk van de drie gevallen wordt kort een adequate relationele operator aangegeven.

**Invoegen** Gegeven is de informatie voor een nieuwe leverancier L4 in een programmagebied W en gevraagd wordt om deze informatie in de database in te voegen. De overeenkomstige opdracht luidt:

- Voeg tupel van W in relatie L.

**Verwijderen** Verwijder het feit dat leverancier L3 artikel A2 kan leveren.

- Verwijder LA tupel met ANO = A2 en LNO = L3.

**Wijzigen** Leverancier L1 is verhuisd naar Amsterdam.

- Wijzig L tupel met LNO = L1 zodanig dat PLAATS := Amsterdam.

## 3.3 HET HIERARCHISCHE DATA MODEL

Figuur 3.3 toont een mogelijke *hiërarchische visie* op de leveranciers-artikelen database. In deze visie worden de gegevens voorgesteld door een eenvoudige boomstructuur, waarbij artikelen hoger geplaatst zijn dan leveranciers. De gebruiker ziet vier individuele bomen of hiërarchieën, een voor ieder artikel. Iedere boom bestaat uit een record voor het artikel tezamen met een verzameling van ondergeschikte records, een voor elke leverancier van dit artikel. Ieder ondergeschikt record bevat de te bestellen hoeveelheid. Merk op dat de verzameling van ondergeschikte records die bij een gegeven artikel hoort leeg kan zijn, namelijk als er geen leverancier voor beschikbaar is (zie bijvoorbeeld artikel A4).

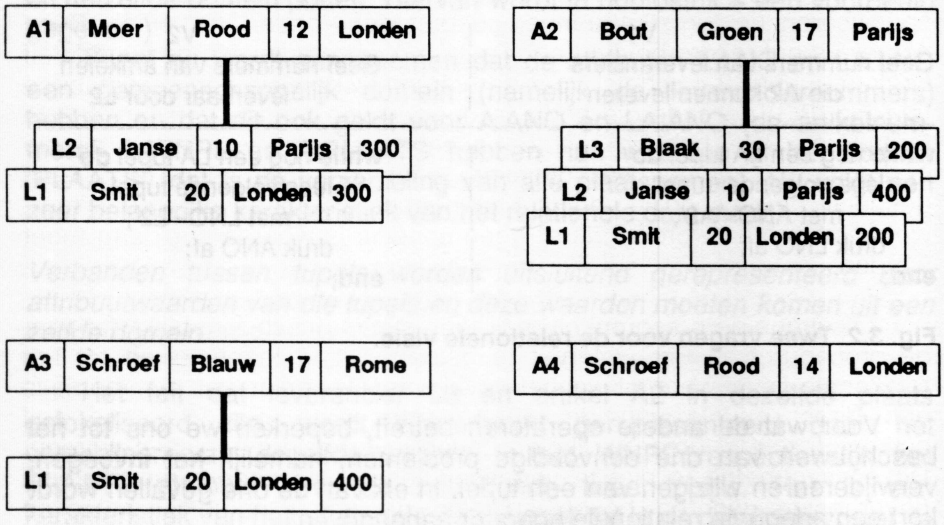



**Fig. 3.3 De hiërarchische visie (artikelen boven leveranciers.)**

Het recordtype aan de top van de boom wordt aangeduid met "wortelrecordtype". Figuur 3.3 is een voorbeeld van een hiërarchische structuur met een wortelrecordtype en *een* ondergeschikt recordtype, hetgeen het meest eenvoudige geval is (als we het 'ontaarde' geval van een hiërarchie bestaande uit alleen het wortelrecordtype even buiten beschouwing laten). In het algemeen mag een wortelrecordtype elk aantal ondergeschikte recordtypen hebben, die op hun beurt ook weer elk aantal ondergeschikte recordtypen mogen hebben, etc. Op deze wijze kunnen bomen met een willekeurige diepte worden gedefinieerd.

In de vorige paragraaf vergeleken we de relationele visie zoals weergegeven in Fig. 3.1 met drie eenvoudige bestanden. Op dezelfde wijze kunnen we de hiërarchische visie van Fig. 3.3 vergelijken met een bestand dat records gerangschikt in vier bomen bevat. De hiërarchische DML, die hieronder zal worden besproken, kan worden gezien als een verzameling operatoren die op deze bestanden werken. Merk echter op dat zo'n bestand complexer van structuur is dan de bestanden die bij relaties horen. In de eerste plaats bevat het records van verschillend type (en niet van hetzelfde type zoals het geval was bij de tabellen); in ons voorbeeld hebben we twee recordtypen namelijk het artikel- en het leverancierrecordtype. In de tweede plaats bevat het bestand ook *verbindingen* die deze records met elkaar verbinden; in ons voorbeeld koppelen de verbindingen de leveranciers van het artikel.

In de hiërarchische visie van een database komt de betekenis van een gegeven record pas volledig tot zijn recht als de context waarin het zich bevindt, wordt meegenomen; het is zelfs zo dat geen enkel ondergeschikt record kan bestaan zonder dat het 'bovengelegen' record bestaat. In de DML moet daarom het analogon van de relationele "lees volgende met" een additionele operand hebben om de context te specificeren (in Fig. 3.4 aangeduid met een "onder" clausule). Deze context is het record boven het gezochte record, tenzij we de wortel zelf zoeken. Figuur 3.4 schetst voor de hiërarchische visie van de database (zie Fig. 3.3) de code die nodig is om de twee vragen van Fig. 3.2 te formuleren. We hebben rechte haken gezet om "volgende", bij die opdrachten waarvoor het onzeker is of tenminste een record de gespecificeerde conditie vervult. Ook hebben we aangenomen dat "met" weggelaten mag worden als er geen conditie is waaraan voldaan moet worden.

Ofschoon de oorspronkelijke vragen symmetrisch zijn, blijkt dat voor de procedures in Fig. 3.4 niet te gelden. Dit in tegenstelling tot het relationele geval waar dit wel het geval was. Dit verlies aan symmetrie is een direct gevolg van de hiërarchische visie (Fig. 3.3) die zelf asymmetrisch is, omdat artikelen boven leveranciers geplaatst worden. Deze asymmetrie is niet te vermijden, omdat de andere mogelijkheid (leveranciers worden boven artikelen geplaatst) natuurlijk hetzelfde nadeel kent. Deze asymmetrie is het grootste nadeel van de

hiërarchische visie; het leidt namelijk tot onnodige complicaties voor de gebruiker. De gebruiker wordt gedwongen om tijd en aandacht te besteden aan het oplossen van problemen die worden geïntroduceerd door de hiërarchische aanpak. Het is duidelijk dat de zaken snel slechter worden als er meer recordtypen een rol gaan spelen en als de hiërarchie complexer wordt. Dit is niet zonder betekenis. Het houdt in dat programma's gecompliceerder worden dan nodig is met als consequentie dat het schrijven, testen en onderhouden van programma's te veel programmeertijd zal eisen.

```
                V1                          |                V2
Geef nummers van leveranciers               |  Geef nummers van artikelen
die A2 kunnen leveren                       |      leverbaar door L2
-----------------------------------------------------------------------------
lees [volgende] artikel met ANO=A2;         |  while nog een artikel do
while nog een leverancier                   |      lees volgende artikel;
      onder dit artikel do                  |      lees [volgende] leverancier
   lees volgende leverancier                |          onder dit artikel
     onder dit artikel;                     |          met LNO=L2;
   druk LNO af;                             |      if found
end;                                        |          then druk ANO af;
                                            |  end;
```

**Fig. 3.4 Twee vragen voor de hiërarchische visie.**

Aan de andere kant zijn hiërarchieën natuurlijk zeer geschikt om echte hiërarchische structuren (een-op-veel associatietypen) uit het object systeem te modelleren. Het leveranciers-artikelen voorbeeld is niet zo'n geval, omdat er een veel-op-veel associatietype tussen leveranciers en artikelen bestaat. Afdelingen en werknemers geven een voorbeeld van een echte hiërarchische structuur (aangenomen dat een werknemer bij precies een afdeling hoort). Maar zelfs in een echte hiërarchische structuur blijft het probleem van de asymmetrie bij opzoekacties bestaan; beschouw bijvoorbeeld de vragen "Vind de werknemers in een gegeven afdeling" en "Vind de afdeling voor een gegeven werknemer". Bovendien hebben hiërarchische structuren vaak de neiging om in de loop van de tijd te evolueren tot veel-op-veel associatietypen!

Als we de bijwerkoperaties beschouwen, dan zien we dat de hiërarchische structuur van Fig. 3.3 enige ongewenste eigenschappen bezit. Er treden namelijk anomalieën (dat wil zeggen afwijkingen, onregelmatigheden) op met betrekking tot de drie bijwerkoperaties (invoegen, verwijderen, wijzigen). Deze anomalieën zijn soms het gevolg van het n:m karakter van de associatie tussen leveranciers en artikelen; sommige anomalieën zouden echter ook optreden als deze associatie van het type 1:n zou zijn (ga dit na!). De moeilijkheden worden

geïllustreerd aan de hand van de drie eenvoudige problemen die ook al aan het eind van de vorige paragraaf werden gebruikt.

**Invoegen** Het is niet mogelijk (zonder een speciaal dummy artikel in te voeren) om gegevens betreffende een nieuwe leverancier, zeg L4, in te voegen zolang deze leverancier geen artikelen kan leveren.

**Verwijderen** De enige manier om de informatie dat een leverancier een artikel in een bepaalde hoeveelheid kan leveren te verwijderen, is het verwijderen van het corresponderende leverancierrecord. Het verwijderen van het enig leverbare artikel van een leverancier houdt dus in, dat we ook alle informatie betreffende de leverancier kwijt zijn (de invoeg- en de verwijderanomalie zijn in feite twee aspecten van hetzelfde probleem). Het verwijderen van de leverbaarheid van artikel A2 door leverancier L3 wordt bijvoorbeeld vertaald in het verwijderen van het L3 record onder artikel A2, hetgeen inhoudt dat alle informatie over L3 nu weg is (L3 komt slechts eenmaal in de database voor).

Een analoog probleem treedt op als we een artikel verwijderen dat toevalligerwijs het enige artikel is dat door een zekere leverancier geleverd kan worden. Dan zullen immers, in overeenstemming met de eis dat geen ondergeschikt leverancierrecord zonder het bovengelegen artikelrecord mag bestaan, alle ondergeschikte records ook worden verwijderd.

**Wijzigen** Als we de gegevens van leverancier L1 moeten wijzigen, bijvoorbeeld omdat hij is verhuisd, dan moeten we de hele database afzoeken naar alle records van leverancier L1 en de vereiste wijziging uitvoeren. Wordt dit in een (of meer) programma('s) niet goed gedaan, dan kan de database in een inconsistente toestand geraken. Dit probleem is een gevolg van gegevensredundantie. Deze redundantie doet zich echter voor op het *gebruikersniveau*, zodat ook de gebruiker verantwoordelijk is voor de handhaving van de consistentie!

## 3.4 HET NETWERK DATA MODEL

Figuur 3.5 toont de *netwerkvisie* van de leveranciers-artikelen database. In deze visie wordt net als in de hiërarchische visie de informatie gerepresenteerd door *records en verbindingen*. Een netwerk is echter een algemenere structuur dan een hiërarchie, omdat een record een meerdere bovengelegen records mag hebben en niet, zoals in het hiërarchische geval, slechts één. Een record mag (natuurlijk) ook een willekeurig aantal ondergeschikte records hebben. De netwerkvisie staat op deze wijze toe veel-op-veel *associatietypen* directer te implementeren dan de hiërarchische visie. Dit wordt in Fig. 3.5 weergegeven. Naast de leverancier- en artikelrecords kennen we ook verbindingrecords. Een verbindingrecord representeert een associatie

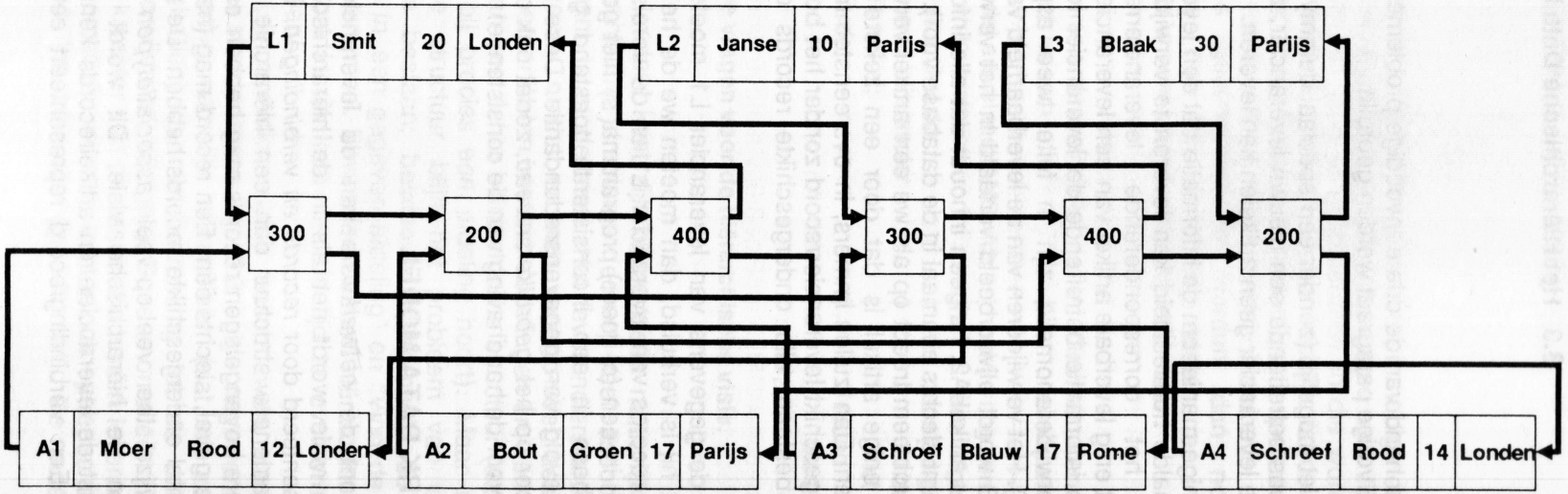



Fig. 3.5 De netwerkvisie.

tussen een leverancier en een artikel en bevat gegevens die deze associatie beschrijven (in het voorbeeld de te bestellen hoeveelheid). Alle verbindingrecords voor een gegeven leverancier worden in een keten geplaatst die begint en eindigt bij die leverancier. Merk op dat de keten op vele wijzen in de opslagstructuur gerealiseerd kan worden (lijststructuren, wijzerketens, etc.). Op dezelfde wijze worden alle verbindingrecords voor een gegeven artikel in een keten geplaatst, die begint en eindigt bij dat artikel. Ieder verbindingrecord zit in precies twee ketens, een leverancierketen en een artikelketen. Zo laat Fig. 3.5 zien, dat leverancier L2 artikel A1 (in de hoeveelheid van 300 stuks) en artikel A2 (in de hoeveelheid van 400 stuks) kan leveren; evenzo toont de figuur, dat artikel A1 door leverancier L1 (in een hoeveelheid 300) en L2 (eveneens in een hoeveelheid van 300) geleverd kan worden. Merk op, dat bij een leverancier vele verbindingrecords horen en bij een verbindingrecord slechts één leverancier, zodat er een een-op-veel associatietype bestaat tussen deze recordtypen, hetgeen inhoudt dat in een netwerksysteem hiërarchieën gemakkelijk gerepresenteerd kunnen worden.

Opnieuw kunnen we de verkregen visie (nu de netwerkvisie) vergelijken met een bestand van records en verbindingen; de interne structuur van dit bestand is nog complexer dan in het hiërarchische geval. Zoals in de netwerk DML van Fig. 3.6 is aangegeven hebben we nu niet alleen een "lees [volgende] onder met" operator nodig, maar ook een "lees boven" operator. Deze operator pakt het eenduidig bepaalde bovengelegen record in een aan te geven keten van een verbindingrecord (feitelijk is de "lees boven" ook nodig in het hiërarchische data model, daar hebben we echter te maken met slechts een keten).

```
                V1                                  |                V2
Geef nummers van leveranciers                       |  Geef nummers van artikelen
 die A2 kunnen leveren                              |       leverbaar door L2
-------------------------------------------------------------------------------------
lees [volgende] A tupel met ANO=A2;                 | lees [volgende] L tupel met LNO=L2;
while nog een verbindingrecord                      | while nog een verbindingrecord
     onder dit artikel do                           |          onder deze leverancier do
   lees volgend verbindingrecord                    |    lees volgend verbindingrecord
      onder dit artikel;                            |          onder deze leverancier;
   lees leverancier boven                           |    lees artikel boven dit
      dit verbindingrecord;                         |          verbindingrecord;
   druk LNO af;                                     |    druk ANO af;
end;                                                | end;
```

**Fig. 3.6 Twee vragen voor de netwerkvisie.**

De netwerkstructuur van Fig. 3.5 is symmetrischer dan de hiërarchische structuur van Fig. 3.3; de symmetrie is weerspiegeld in de twee procedures van Fig. 3.6. Deze procedures zijn echter veel complexer dan in de relationele situatie. Voor vraag V1 geldt zelfs dat de hiërarchische oplossing eenvoudiger is, zie Fig. 3.4 (dat wil zeggen dat symmetrie ook niet alles is).

Een complicatie die in Fig. 3.6 niet naar voren komt, blijkt bij vragen als "Vind de hoeveelheid waarin leverancier L2 artikel A2 kan leveren". Om een dergelijke vraag te beantwoorden, moeten we het (unieke) verbindingrecord te pakken zien te krijgen dat zowel op de keten van A2 als op die van L2 ligt. Het probleem is dat er twee strategieën zijn om dat record te lokaliseren. De ene strategie start bij leverancier L2 en loopt de keten verbindingrecords af om het record te vinden dat met A2 verbonden is. De andere strategie begint juist bij het A2 record en loopt zijn verbindingrecords af om het verbindingrecord te vinden dat met L2 verbonden is. Het is voor de gebruiker in het algemeen moeilijk om uit te maken welke strategie gevolgd moet worden. De keuze kan zeer significant zijn (in snelheid gemeten).

Gelijke opmerkingen kunnen voor bijwerkoperaties gemaakt worden. We constateren dat de anomalieën van de vorige paragraaf hier niet optreden. De benodigde programmering is echter niet altijd van de gewenste duidelijkheid. Bij het verwijderen, bijvoorbeeld, komen we hetzelfde probleem tegen als in de vorige paragraaf is beschreven.

**Invoegen** Om de gegevens van een nieuwe leverancier in te voegen, L4 bijvoorbeeld, maken we een nieuw leverancierrecord. In het begin zijn er voor deze leverancier geen verbindingrecords, aangezien L4 dan nog geen artikelen kan leveren. De keten van verbindingrecords bevat dan slechts een wijzer, namelijk van het leverancierrecord naar zichzelf.

**Verwijderen** Volgens Fig. 3.5 kan A2 door L3 geleverd worden. Om dit ongedaan te maken verwijderen we het verbindingrecord dat de records van A2 en L3 verbindt. De twee betrokken ketens zullen op de juiste wijze (automatisch) worden aangepast.

**Wijzigen** Als leverancier L1 verhuist, hoeven we niet de hele database af te zoeken om alle L1 records op te sporen. Dit omdat er geen gegevensredundantie is: de plaats waar L1 woont komt slechts eenmaal in de database voor.

We stellen daarom dat het belangrijkste nadeel van de netwerkvisie de overmatige complexiteit in zowel gegevensstructuren als in de bijbehorende DML is. De bron van deze complexiteit is gelegen in de informatie-dragende constructies die door het netwerkmodel worden ondersteund (naast de noodzakelijke records ook de verbindingen). In

het algemeen kan men stellen dat hoe groter het aantal gegevensstructuren is des te meer operatoren er nodig zijn om deze te behandelen en dus des te complexer de DML wordt. Hierop komen we later nog terug.

## 3.5 HOGERE NIVEAU OPERATOREN

Tot nu toe hebben we steeds stilzwijgend aangenomen dat, net als in de programmering van de conventionele bestanden, alle operatoren in principe een tupel (record) als operand hebben. Echter, veel problemen worden het natuurlijkst uitgedrukt in termen van verzamelingen en niet in termen van tupels (beschouw bijvoorbeeld de vragen V1 en V2 van Fig. 3.2). In deze paragraaf introduceren we een krachtige taal, waarvan de operatoren in staat zijn om in één keer met verzamelingen van tupels in plaats van met een enkel tupel te manipuleren.

We beschouwen eerst enige vragen die betrekking hebben op de relationele visie van Fig. 3.1.

**3.5.1** Vind WNPL van leverancier L1.

```
          L                                  RESULTAAT
LNO  LNAAM    STATUS  WNPL                    WNPL
---------------------------                   -----
 L1  Smit      20     Londen                  Londen
 L2  Janse     10     Parijs
 ...............
```

Het antwoord is Londen. Preciezer, het antwoord is een relatie met één attribuut (namelijk WNPL, domein is PLAATS) en één tupel (namelijk "Londen").

**3.5.2** Vind LNO en STATUS van de Parijse leveranciers.

```
          L                                  RESULTAAT
LNO  LNAAM    STATUS  WNPL                  LNO  STATUS
---------------------------                 -----------
 L2  Janse     10     Parijs                 L2    10
 L3  Blaak     30     Parijs                 L3    30
 ...............
```

Het resultaat is opnieuw een relatie. Deze keer met twee attributen en twee tupels.

**3.5.3** Vind ANAAM van artikelen die leverancier L1 kan leveren.

```
    LA      +        A          ->          RESULTAAT
LNO   ANO        ANO  ANAAM                 ANAAM
---------        ----------                 ------
 L1    A1         A1  Moer                   Moer
 L1    A2         A2  Bout                   Bout
 L1    A3         A3  Schroef                Schroef
```

Opnieuw is het resultaat een relatie. In feite kan het resultaat van elke vraag worden beschouwd als een relatie. In dit voorbeeld wordt de resultaatrelatie verkregen door twee relaties te onderzoeken, namelijk LA en A.

**3.5.4** Geef voor ieder leverbaar artikel het artikelnummer (ANO) en de woonplaats (WNPL) van de betreffende leverancier.

```
    LA      +        L          ->          RESULTAAT
LNO   ANO        LNO  WNPL                  ANO  WNPL
---------        -----------                ----------
 L1    A1         L1  Londen                 A1  Londen
 L1    A2         L2  Parijs                 A2  Londen
 L1    A3         L3  Parijs                 A3  Londen
 L2    A1                                    A1  Parijs
 L2    A2                                    A2  Parijs
 L3    A2                  (Verwijderd)      A2  Parijs
```

In dit laatste voorbeeld worden opnieuw twee relaties onderzocht om het resultaat te bepalen. Bovendien wordt het resultaat samengesteld uit informatie van de twee onderzochte relaties. Merk op dat van het uiteindelijke resultaat een (redundant) duplicaattupel is verwijderd. De reden is dat, wiskundig gesproken, een relatie een verzameling van tupels is en een verzameling mag geen duplicaatelementen bevatten.

Het resultaat van een vraag is een relatie die op de een of andere wijze van de relaties in de database is afgeleid; elk aantal relaties mag betrokken zijn bij het vormen van het resultaat. Dit kan zich openbaren in het formuleren van zowel de selectieconditie waaraan bijdragende tupels moeten voldoen, als de specificatie van de uit te voeren de attributen. Met andere woorden, *het verwerken van een vraag houdt de constructie van een relatie in.* Dit feit erkend hebbende kunnen we een aantal operatoren voor het construeren van relaties definiëren. We zullen in het kort drie van zulke operatoren bespreken: SELECT, PROJECT en JOIN.

De SELECT operator construeert een nieuwe relatie door een *horizontale* deelverzameling van een bestaande relatie te nemen. Deze

deelverzameling bevat alle tupels van een bestaande relatie, die aan een bepaalde conditie voldoen. De PROJECT operator daarentegen, vormt een *verticale* deelverzameling van een bestaande relatie door de gespecificeerde attributen er uit te lichten en eventuele redundante duplicaattupels te verwijderen. Door van deze twee operatoren gebruik te maken kunnen we de eerste twee voorbeelden al programmeren.

**3.5.1** Vind WNPL van leverancier L1.

*Stap 1.*
```
SELECT L  WHERE LNO = 'L1'  GIVING TEMP
```

Deze stap geeft ons de volgende relatie (bestaande uit een tupel, namelijk het 'L1' tupel).

```
          TEMP
LNO   LNAAM   STATUS   WNPL
-----------------------------
 L1   Smit      20      Londen
```

*Stap 2.*
```
PROJECT  TEMP  OVER WNPL GIVING RESULTAAT
```

Deze stap maakt een kopie van het WNPL attribuut van TEMP en kent het resultaat toe aan RESULTAAT.

**3.5.2** Vind LNO en STATUS van de Parijse leveranciers.

```
SELECT   L   WHERE WNPL = 'PARIJS'  GIVING TEMP
PROJECT  TEMP  OVER (LNO,STATUS)    GIVING RESULTAAT
```

Dit voorbeeld lijkt sterk op het vorige.

De twee overblijvende voorbeelden (waarbij twee relaties een rol spelen) eisen het gebruik van de JOIN operator. Als twee relaties een attribuut hebben dat is gedefinieerd over hetzelfde domein, dan kunnen ze over deze twee attributen worden verbonden. Het resultaat van de verbinding is een nieuwe, 'bredere' relatie waarin ieder tupel wordt gevormd door twee tupels aaneen te voegen, één tupel van beide relaties. Dit aaneenschakelen mag niet geschieden voor willekeurige tupels van de twee relaties, nee, de twee tupels moeten een gelijke waarde voor de betreffende twee attributen hebben. Zo mogen de relaties L en A aaneen worden gevoegd over de attributen WNPL en PLAATS, omdat deze twee attributen over het domein PLAATS zijn gedefinieerd. Het resultaat is te zien in Fig. 3.7. Merk op dat als een tupel in een relatie geen tegenhanger heeft in de andere relatie, dat dan het tupel eenvoudig niet in het resultaat voorkomt. Zie bijvoorbeeld A3 (opgeslagen in Rome) dat niet in het resultaat voorkomt.

De relatie in Fig. 3.7 kent twee attributen die voor alle tupels dezelfde waarde hebben, namelijk WNPL en PLAATS. Dit volgt natuurlijk rechtstreeks uit de definitie van de JOIN operatie. Feitelijk komt deze definitie overeen met een van de vele mogelijke JOIN operatoren die in zwang zijn, namelijk die JOIN operator waarbij de 'aaneenschakel-conditie' is gebaseerd op *gelijkheid* van de waarden van de twee JOIN-attributen. Deze JOIN wordt gewoonlijk de 'equijoin' genoemd. Het is ook mogelijk om een 'groter-dan-join' en een 'niet-gelijk-aan-join' etc. te definiëren. Het is wel zo dat de equijoin verreweg de belangrijkste is. Het is natuurlijk mogelijk om een van de twee attributen die gelijke waarden opleveren via een PROJECT operatie weg te werken; een equijoin waarbij een van de twee identieke attributen (wat waarden betreft) is weggewerkt, noemt men een *natuurlijke join*. Deze natuurlijke join is belangrijk in de context van normalisatie, zie hoofdstuk 6.

| LNO | LNAAM | STATUS | WNPL | ANO | ANAAM | KLEUR | GEW | PLAATS |
|---|---|---|---|---|---|---|---|---|
| L1 | Smit | 20 | Londen | A1 | Moer | Rood | 12 | Londen |
| L1 | Smit | 20 | Londen | A4 | Schroef | Rood | 14 | Londen |
| L2 | Janse | 10 | Parijs | A2 | Bout | Groen | 17 | Parijs |
| L3 | Blaak | 30 | Parijs | A2 | Bout | Groen | 17 | Parijs |

Fig. 3.7 JOIN van L en A over WNPL en PLAATS.

Nu kunnen we de twee andere voorbeelden programmeren.

**3.5.3** Vind ANAAM van de artikelen die door leverancier L1 geleverd kunnen worden.

```
SELECT   LA WHERE LNO = 'L1'     GIVING TEMP1
JOIN   TEMP1 AND A OVER (ANO,ANO)    GIVING TEMP2
PROJECT   TEMP2 OVER ANAAM   GIVING RESULTAAT
```

**3.5.4** Geef voor ieder leverbaar artikel het ANO en de WNPL van de betreffende leverancier.

```
JOIN   LA AND L OVER (LNO,LNO) GIVING TEMP
PROJECT TEMP OVER (ANO,WNPL) GIVING RESULTAAT
```

In dit voorbeeld zorgt de definitie van PROJECT ervoor dat er geen duplicaat tupels in het resultaat aanwezig zullen zijn.

De operatoren SELECT, PROJECT en JOIN vormen tezamen met andere operatoren die op verzamelingen opereren (UNION, DIFFERENCE, etc) de zogenaamde *relationele algebra*. Elke operator van de relationele algebra pakt een of twee relaties als operanden en

produceert een nieuwe relatie als resultaat. Zoals we hebben aangetoond is het mogelijk (tenminste voor het formuleren van vragen) om een DML te definiëren met operatoren die op verzamelingen werken. Ten aanzien van het bijwerken van relationele databases beperken we ons tot de bewering, dat het ook daar mogelijk is om operatoren te definiëren die op verzamelingen van tupels werken. Voor de relationele systemen is het dus mogelijk om een DML op verzamelingsniveau te definiëren.

Hoe staat het nu met de hiërarchische en netwerkbenadering? Het zou misleidend zijn om te beweren dat talen op verzamelingsniveau voor deze benaderingen niet te definiëren zouden zijn. Echter, bij deze benaderingen kan informatie op meer dan een manier gerepresenteerd worden, en dat leidt tot meer dan één verzameling van operatoren in de DML. Zonder in details te treden beweren we dat een hiërarchische of netwerktaal die op verzamelingsniveau moet opereren, complexer is dan die van de relationele benadering.

## OPGAVEN

Een database bevat gegevens over personen en bekwaamheden. Op een zeker moment zijn de volgende personen (met hun bekwaamheden) in de database opgenomen.

| *Persoon* | *Bekwaamheden* |
|---|---|
| Arthur | Programmeren |
| Wil | Ontwerpen en Programmeren |
| Charlie | Analyseren, Ontwerpen en Programmeren |
| David | Ontwerpen en Analyseren |

Voor iedere persoon bevat de database enige gegevens zoals het adres. Voor elke bekwaamheid is in de database naast de naam ook een code en andere informatie opgenomen. Een bekwaamheid wordt verworven als een bepaalde cursus gevolgd is. De identificatie van deze cursus moet bij de bekwaamheid zijn genoteerd. De database bevat ook de datum waarop een persoon zich een bepaalde bekwaamheid eigen heeft gemaakt.

**3.1** Geef voor deze gegevens de relationele structuur.

**3.2** Geef *twee* hiërarchische structuren voor deze gegevens.

**3.3** Geef een netwerkstructuur voor deze gegevens.

**3.4** Geef voor de drie bovenstaande situaties een schets van een procedure die de namen van de personen bepaalt, die de bekwaamheid Analyseren bezitten.
In de relationele situatie moeten oplossingen worden gegeven voor beide taalniveaus, dat wil zeggen voor het 'tuple-at-a-time' (zie paragraaf 3.2-3.4) en voor het 'set-at-a-time' niveau (zie paragraaf 3.5).

# 4
# Relationele Databases

## 4.1 INLEIDING

In hoofdstuk 3 is het relationele data model informeel ingevoerd. Daarbij kwamen de begrippen *domein*, *attribuut*, *tupel* en *relatie* aan de orde. Deze begrippen worden in Fig. 4.1 nogmaals herhaald voor de leveranciers relatie L. Bovendien komt in Fig. 4.1 het nieuwe begrip *primaire sleutel* voor. In de nu volgende paragrafen zal een meer formele definitie van deze begrippen worden gegeven.

## 4.2 DOMEINEN

De kleinste gegevenseenheid in het relationele data model is een individuele gegevenswaarde, zoals een leveranciernummer of een te bestellen hoeveelheid. Zulke waarden worden *atomair* (ondeelbaar) genoemd, hetgeen inhoudt dat ze voor het data model niet meer op te splitsen zijn. (Natuurlijk is een waarde bijvoorbeeld opgebouwd uit cijfers, maar voor het data model vormen deze cijfers een geheel, namelijk de reeds genoemde waarde.) Een *domein* (domain) is een verzameling van ondeelbare waarden die van hetzelfde type zijn. Zo is bijvoorbeeld het domein van het attribuut L.LNO de verzameling van alle *mogelijke* leveranciernummers, het domein van het attribuut LA.HOEV de verzameling van alle natuurlijke getallen groter dan 0 en kleiner dan bijvoorbeeld 10.000. Domeinen zijn dus verzamelingen van waarden, waaruit de attributen hun actuele waarden putten. Merk op, dat op een willekeurig moment er waarden mogen zijn (en in de praktijk meestal zullen zijn) die wel in het domein voorkomen, maar die niet worden aangenomen door een van de attributen die op dat domein zijn gedefinieerd. Zo kan bijvoorbeeld L8 een waarde zijn in het domein van

de leveranciernummers, dat wil zeggen L8 zou het nummer van een leverancier kunnen zijn, maar toch komt L8 niet voor in Fig. 4.1.

Domeinen hebben een zekere operationele betekenis: als twee attributen hun waarden uit hetzelfde domein putten, dan zijn vergelijkingen tussen deze twee attributen waarschijnlijk zinvol. Omgekeerd, als twee attributen hun waarden uit verschillende domeinen halen, dan zijn vergelijkingen van die twee attributen waarschijnlijk niet zinvol. Zo zal de vergelijking "A.ANO = LA.ANO" (A.ANO geeft het ANO attribuut uit relatie A weer) waarschijnlijk zinvol zijn, omdat beide attributen op het domein 'artikelnummer' gedefinieerd zijn. Daarentegen zal de vergelijking "A.GEW = LA.HOEV" waarschijnlijk niet zinvol zijn, omdat het getal waarin een gewicht wordt uitgedrukt een andere betekenis heeft dan het getal waarin een hoeveelheid wordt uitgedrukt.

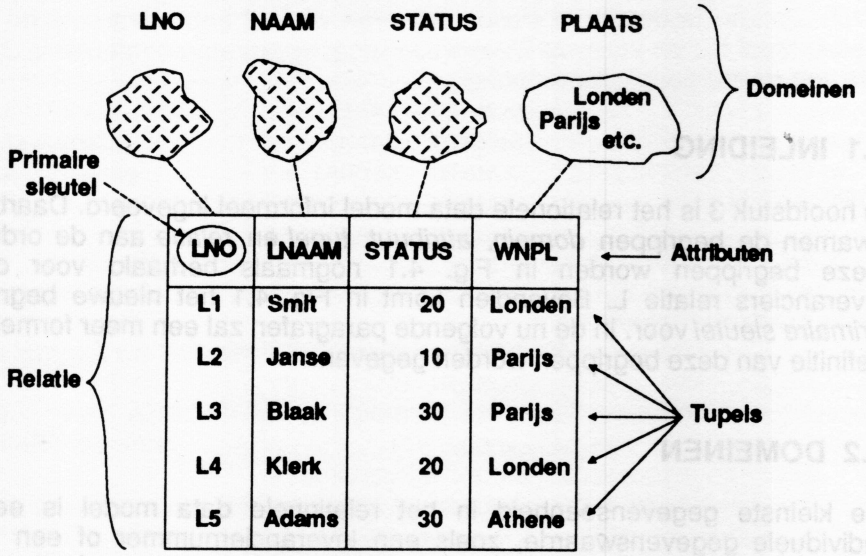



**Fig. 4.1 De leveranciers relatie L.**

Merk op dat domeinen in de eerste plaats conceptueel van aard zijn. Ze kunnen, maar hoeven niet expliciet als een actuele verzameling waarden in de database opgeslagen te zijn; meestal zullen ze niet expliciet opgeslagen zijn. Ze moeten daarentegen wel opgenomen zijn in de databasedefinitie, en iedere definitie van een attribuut moet een verwijzing naar het corresponderende domein bevatten, zodat het systeem weet welke attributen vergelijkbaar zijn en welke niet. Figuur 4.2 laat zien hoe de leveranciers-artikelen database eruit ziet (op *schema niveau*) als de domeinen worden meegenomen.

```
DOMAIN      LNO       CHAR(5)
DOMAIN      NAAM      CHAR(20)
DOMAIN      STATUS    SMALLINT
DOMAIN      PLAATS    CHAR(15)
DOMAIN      ANO       CHAR(6)
DOMAIN      KLEUR     CHAR(6)
DOMAIN      GEWICHT   SMALLINT
DOMAIN      HOEVHD    INTEGER

RELATION        L
      ( LNO         DOMAIN (LNO),
        LNAAM       DOMAIN (NAAM),
        STATUS      DOMAIN (STATUS),
        WNPL        DOMAIN (PLAATS) ) ,
      PRIMARY KEY   LNO

RELATION        A
      ( ANO         DOMAIN (ANO),
        ANAAM       DOMAIN (NAAM),
        KLEUR       DOMAIN (KLEUR),
        GEW         DOMAIN (GEWICHT),
        PLAATS      DOMAIN (PLAATS) ) ,
      PRIMARY KEY   ANO

RELATION        LA
      ( LNO         DOMAIN (LNO),
        ANO         DOMAIN (ANO),
        HOEV        DOMAIN (HOEVHD) ) ,
      PRIMARY KEY   (LNO,ANO) ,
      FOREIGN KEY   LNO IDENTIFIES L ,
      FOREIGN KEY   ANO IDENTIFIES A
```

**Fig. 4.2 De leveranciers-artikelen database-definitie.**

Per domein wordt het datatype weergegeven. Zo staat bijvoorbeeld SMALLINT voor 'small integer'; de andere typen worden bekend verondersteld. Op de PRIMARY KEY en FOREIGN KEY clausules wordt in paragraaf 4.4 teruggekomen.

Een gegeven attribuut mag dezelfde, maar ook een andere naam hebben als het domein waaruit de waarden worden geput. Vanzelfsprekend is een andere naam nodig als dubbelzinnigheid dreigt; het kan namelijk voorkomen dat in een relatie meer dan een attribuut aan een en hetzelfde domein waarden ontleent (in de volgende paragraaf wordt hiervan een voorbeeld gegeven).

Er zijn waarden zoals datum, die een samengesteld karakter hebben, in dit geval een jaar-maand-dag waarde. Deze waarden kunnen tezamen een domein vormen en als atomair (ondeelbaar) worden

beschouwd als het database systeem de er in aanwezige structuur niet hoeft te kennen.

## 4.3 RELATIES

We kunnen nu het begrip "relatie" definiëren. Een relatie op de domeinen D1,D2,...,Dn (die niet noodzakelijk alle verschillend behoeven te zijn) bestaat uit een *intensie* (intension: beschrijving van het type van de relatie) en een *extensie* (extension: verzameling tupels van een relatie).

- De intensie bestaat uit een verzameling attributen A1,A2,...,An zodanig dat het attribuut Ai correspondeert met domein Di (i = 1,2,...,n)

- De extensie bestaat uit een verzameling tupels, waarbij ieder tupel op zijn beurt bestaat uit een verzameling paren (Ai,wi), één paar voor elk attribuut in de intensie (dus i = 1,2,...,n). Voor elk paar (Ai,wi) in de extensie moet gelden dat wi een waarde is uit het unieke domein Di dat met attribuut Ai geassocieerd is.

In het algemeen zal de extensie van een relatie in de loop van de tijd veranderen: er kunnen tupels bijkomen of afgaan. Bovendien kunnen tupels van inhoud veranderen, waardoor ook de extensie wijzigt.

Hoe voldoet de leveranciers relatie L van Fig. 4.1 aan deze definitie? De onderliggende domeinen zijn de domeinen van leveranciernummers (LNO), namen (NAAM), statuswaarden (STATUS) en plaatsnamen (PLAATS). De intensie van L bestaat uit de attributen LNO (met als onderliggend domein LNO), LNAAM (domein NAAM), STATUS (domein STATUS) en WNPL (domein PLAATS). De extensie van L bestaat uit de verzameling van tupels. In Fig. 4.1 zijn dat er vijf, maar dit aantal zal in de loop van de tijd variëren als invoegingen en verwijderingen met betrekking tot L plaatsvinden. Elk tupel bestaat uit een verzameling van vier attribuut-waarde paren, een paar voor elk van de vier attributen in de intensie. Zo bestaat het tupel van leverancier L1 bijvoorbeeld uit de paren:

```
( LNO    : 'L1'      )
( LNAAM  : 'Smit'    )
( STATUS:  20        )
( WNPL   : 'Londen'  )
```

In een informele representatie van een relatie worden de domeinen vaak weggelaten. Natuurlijk moet iedere attribuutwaarde komen uit het onderliggende domein. Zo zal de waarde L1 bijvoorbeeld moeten

voorkomen in het domein LNO. Dus L als geschetst in Fig. 4.1. is inderdaad een relatie die voldoet aan de definitie.

De waarde n (het aantal attributen in de relatie) wordt de *graad* van de relatie genoemd. Een relatie van graad één heet *unair*, een relatie van graad twee *binair*, van graad drie *ternair*, en een relatie van graad n is een *n-aire* relatie. In de leveranciers-artikelen database hebben de relaties L, A en LA respectievelijk de graad 4, 5 en 3. Het aantal tupels in de relatie wordt de *cardinaliteit* van de relatie genoemd. In het voorbeeld van Fig. 3.1 zijn de cardinaliteiten van de relaties L, A en LA respectievelijk 3, 4 en 6. De cardinaliteit van een relatie kan in de tijd veranderen, de graad (normaal gesproken) niet.

Merk op dat de onderliggende domeinen van een relatie "niet noodzakelijkerwijs allemaal verschillend" zijn. Een voorbeeld hiervan wordt gegeven in Fig. 4.3. Als hetzelfde domein meer dan eens voorkomt in een relatie, dan is het niet mogelijk om alle attributen van de relatie de naam van het onderliggende domein te geven (dan treedt immers dubbelzinnigheid op). Figuur 4.3 laat de aanbevolen aanpak voor zo'n situatie zien: genereer verschillende attribuutnamen door de naam van het gemeenschappelijk domein te voorzien van een voorvoegsel dat uniek is en dat de rol weergeeft die dat domein bij ieder optreden speelt. In Fig. 4.3 geeft het eerste tupel van ARTIKEL_STRUCT weer, dat er in een A1 artikel 2 stuks van artikel A2 zitten.

ARTIKEL_STRUCT

| HOOFD_ANO | ONDER_ANO | HOEV |
|---|---|---|
| A1 | A2 | 2 |
| A1 | A4 | 4 |
| A5 | A3 | 1 |
| A3 | A6 | 3 |
| A6 | A1 | 9 |
| A5 | A6 | 8 |
| A2 | A4 | 3 |

**Fig. 4.3 De ARTIKEL_STRUCT relatie.**

### Eigenschappen van relaties

Relaties bezitten belangrijke eigenschappen. We geven die eigenschappen eerst kort weer, waarna een meer gedetailleerde uiteenzetting zal volgen. Voor een gegeven relatie geldt:

- er mogen geen duplicaattupels zijn;
- de tupels zijn ongeordend;

- de attributen zijn ongeordend;
- de attribuutwaarden zijn atomair.

*1. Er zijn geen duplicaattupels*

Deze eigenschap volgt uit het feit dat de extensie van een relatie een wiskundige verzameling is (namelijk een verzameling van tupels) en een verzameling kent per definitie geen duplicaten.

Omdat tupels uniek zijn, bezit tenminste de combinatie van alle attributen van de relatie de *uniciteitseigenschap*. Een combinatie van attributen heeft deze eigenschap als de waarden van deze attributen een tupel uniek bepalen, dat wil zeggen, identificeren. Normaliter is het niet nodig om alle attributen er bij te betrekken; een kleinere combinatie is gewoonlijk voldoende. In paragraaf 4.4, bij de definitie van primaire sleutel, zal blijken dat de primaire sleutel geen attributen bevat die overbodig zijn vanuit het oogpunt van de unieke identificatie. Zo is de combinatie (LNO, WNPL), ofschoon "uniek", niet de primaire sleutel van relatie L, omdat WNPL kan worden weggelaten zonder dat de eigenschap dat een tupel uniek wordt geïdentificeerd verloren gaat. Merk echter op, dat de primaire sleutel best samengesteld kan zijn, dat wil zeggen meer dan een attribuut kan bevatten. Denk bijvoorbeeld aan relatie LA die als sleutel (LNO,ANO) heeft.

*2. Tupels zijn ongeordend*

Deze eigenschap volgt ook direct uit het feit dat de extensie van een relatie een wiskundige verzameling is. Verzamelingen zijn nu eenmaal niet geordend. In Fig. 4.1 bijvoorbeeld, hadden de tupels van relatie L net zo goed in omgekeerde volgorde getoond kunnen worden (of een andere volgorde); het zou nog steeds dezelfde relatie zijn geweest.

Ter zijde zij opgemerkt dat het enigszins ongelukkig is dat relaties op papier in de vorm van tabellen worden gerepresenteerd, omdat deze representatie suggereert dat er een ordening is. Het moet duidelijk zijn dat een tabel een concrete representatie is van een abstract begrip (namelijk een relatie) en dat sommige aspecten van de concrete representatie een foutief beeld geven. Ondanks dit zwakke punt, is het in feite juist een sterk punt van het relationele data model dat haar gegevensstructuren een concrete representatie hebben die zo vertrouwd is en die zo gemakkelijk te begrijpen is. Deze representatie vergemakkelijkt het redeneren over de effecten van de verschillende operaties op die structuren zeer. Als zodanig draagt de representatie bij tot het gebruiksgemak van relationele systemen.

*3. Attributen zijn ongeordend*

Deze eigenschap volgt uit het feit dat de intensie van een relatie ook als een verzameling is gedefinieerd (namelijk als een verzameling attributen). Strikt gesproken bestaat er niet zoiets als het eerste, tweede of derde attribuut van relatie L; in plaats daarvan is er een STATUS attribuut dat altijd wordt geadresseerd via de naam en niet via de positie. Een gevolg hiervan is dat de mogelijkheden om fouten te maken en onduidelijk te programmeren, worden verkleind. Zo is er bijvoorbeeld geen manier om het systeem te ondermijnen door op de een of andere wijze van het ene attribuut naar het andere te springen. Dit staat in scherp contrast met de situatie die men in veel programmeersystemen tegenkomt, namelijk dat het mogelijk is om het fysiek naast elkaar liggen van logisch verschillende elementen op subversieve manieren uit te buiten. Dit is in hoge mate strijdig met de eis van fysieke gegevensonafhankelijkheid en, in meer algemene zin, met goede programmeerstijl.

Zoals reeds onder eigenschap 2 is aangegeven, wordt door een tabelrepresentatie van een relatie gesuggereerd dat de attributen geordend zijn. Nogmaals, dit is niet waar: de attributen van een relatie zijn ongeordend.

*4. Alle attribuutwaarden zijn atomair*

Deze eigenschap is een gevolg van het feit dat de onderliggende domeinen alleen atomaire waarden bevatten. We kunnen de eigenschap informeel anders formuleren, namelijk als volgt: op iedere rij-kolom positie in een tabelrepresentatie van een relatie mag slechts *een* waarde voorkomen, nooit een verzameling van waarden. Of weer anders geformuleerd: *relaties bevatten geen repeterende groepen* (repeating groups, zoals bijvoorbeeld attribuut FH in Fig. 4.4). Een relatie die eigenschap 4 bezit, is in de eerste normaalvorm (1NV) of anders gezegd, is *genormaliseerd*.

Aangezien relaties (in de context van het relationele data model) eigenschap 4 bezitten, is iedere relatie automatisch in 1NV. Het punt is echter, dat een *wiskundige relatie niet noodzakelijk genormaliseerd* behoeft te zijn. Beschouw de wiskundige relatie VOOR zoals die in tabelvorm in Fig. 4.4 is weergegeven. VOOR beschrijft per werknemer de functies (met aanvangsdatum) die hij in zijn carrière bij het bedrijf heeft vervuld en kent twee attributen, namelijk werknemernummer (WNO) en functiehistorie (FH). Zoals gezegd, wiskundig gezien is VOOR een relatie van graad 2; in de context van het relationele data model is VOOR echter niet een relatie. Het blijkt dat een van de onderliggende domeinen, namelijk het domein horend bij het attribuut FH, waarden bevat die zelf ook weer relaties zijn. Attribuut FH is gedefinieerd op een relatie-waardig domein, waarvan de elementen binaire relaties zijn. Deze binaire relaties zelf hebben twee attributen, namelijk functiecode (FC) en

aanvangsdatum (AD), die gedefinieerd zijn op domeinen die atomaire waarden bevatten. Het relationele data model staat geen relatie-waardige domeinen toe.

VOOR

| WNO | FH | |
|---|---|---|
| | FC | AD |
| W1 | F1 | 01-01-81 |
| | F2 | 15-03-83 |
| | F5 | 01-06-84 |
| | F9 | 01-09-86 |
| W2 | F3 | 01-04-82 |
| | F7 | 01-01-84 |
| | F9 | 01-11-85 |
| W3 | F2 | 01-11-84 |
| | F6 | 01-01-86 |
| | F8 | 01-04-87 |

NA

| WNO | FC | AD |
|---|---|---|
| W1 | F1 | 01-01-81 |
| W1 | F2 | 15-03-83 |
| W1 | F5 | 01-06-84 |
| W1 | F9 | 01-09-86 |
| W2 | F3 | 01-04-82 |
| W2 | F7 | 01-01-84 |
| W2 | F9 | 01-11-85 |
| W3 | F2 | 01-11-84 |
| W3 | F6 | 01-01-86 |
| W3 | F8 | 01-04-87 |

**Fig. 4.4 Een voorbeeld van normalisatie.**

Een relatie zoals VOOR moet in het relationele model worden vervangen door een semantisch equivalente, doch genormaliseerde relatie. De relatie NA, die van VOOR is afgeleid, is zo'n relatie. De graad van NA is drie en alle domeinen bevatten atomaire waarden (zoals vereist in het relationele data model).

De relatie VOOR is *ongenormaliseerd* en het proces om van de relatie VOOR de relatie NA te maken, wordt *normaliseren* genoemd. Zoals het voorbeeld aantoont, is het niet moeilijk om een ongenormaliseerde (wiskundige) relatie om te zetten in een genormaliseerde relatie. Het normalisatieproces is echter veel uitgebreider dan hier aangegeven. Daar wordt echter in hoofdstuk 6 nog uitgebreid op teruggekomen.

Het uitsluiten van ongenormaliseerde relaties in het relationele model heeft de volgende reden. Een genormaliseerde relatie heeft, wiskundig gesproken, een veel eenvoudigere structuur dan een ongenormaliseerde. Het gevolg is, dat de bijbehorende operatoren ook eenvoudiger en kleiner in aantal zijn. Beschouw bijvoorbeeld de volgende twee transacties:

T1. Voeg een nieuwe functie voor werknemer W5 toe met functiecode F7 en aanvangsdatum 01-01-87.

T2. Voeg een nieuwe functie voor werknemer W3 toe met functiecode F7 en aanvangsdatum 01-01-87.

Voor relatie NA is er geen verschil tussen deze twee transacties; allebei moeten ze een tupel aan de relatie toevoegen. Met de (wiskundige) relatie VOOR echter, moet T1 nog steeds een tupel invoegen, maar moet T2 heel iets anders doen, namelijk het toevoegen van een waarde aan een verzameling bestaande waarden in een bestaand tupel. Dat wil zeggen dat er twee "INVOEG" operaties nodig zijn om ongenormaliseerde relaties te ondersteunen. Om dezelfde reden hebben we twee LEES, VERWIJDER, etc. operaties. Merk op, dat dit niet alleen geldt voor manipulatieopdrachten, maar voor *alle* systeemoperaties. Zo zullen extra operaties nodig zijn op het gebied van beveiliging, integriteitsbewaking, etc. Al met al wordt het zeer complex.

## 4.4 SLEUTELS

Het integriteitsdeel (voor dit begrip, zie paragraaf 2.7) van het relationele data model bestaat uit twee algemene integriteitsregels, namelijk de *entiteit-integriteitsregel* en de *referentie-integriteitsregel.* Deze regels zijn algemeen in die zin, dat ze van toepassing zijn op iedere database waarvan beweerd wordt dat hij opgezet is overeenkomstig de voorschriften van het relationele data model. De integriteitsregels zullen besproken worden in paragraaf 4.5. Natuurlijk zal iedere database, naast deze twee, nog additionele integriteitsregels kennen. Zo zou bijvoorbeeld de leveranciers-artikelen database een regel kunnen hebben, die zegt dat het gewicht van een artikel minstens 13 moet zijn als de kleur van dat artikel zwart is (dit kan gelden als kleuren iets zeggen over gewichten). Deze specifieke regels vallen echter buiten de algemene eigenschappen van het relationele data model.

De twee algemene regels betreffen respectievelijk *primaire sleutels* (primary keys) en *vreemde sleutels* (foreign keys). We onderzoeken daarom eerst deze twee begrippen.

**4.4.1 Primaire sleutels.** Primaire sleutels vormen eigenlijk een speciaal geval van een meer algemeen begrip, namelijk van *kandidaatsleutels* (candidate keys). Een kandidaatsleutel is een attribuut of een combinatie van attributen met de eigenschap dat hij tupels uniek identificeert. In de vorige paragraaf werd dit de uniciteitseigenschap genoemd. Aangezien tupels representaties van entiteiten zijn, identificeert een kandidaatsleutel dus de entiteiten waarvan de tupels representaties zijn. Omdat in een relatie ieder tupel uniek is (er zijn geen duplicaten), bestaat er altijd een kandidaatsleutel. De meeste relaties hebben *precies één* kandidaatsleutel, maar sommige relaties hebben er twee of meer. Per relatie kiezen we een primaire sleutel; de andere kandidaatsleutels (als ze er zijn) worden dan zogenaamde alternatieve sleutels.

We gaan nu een en ander preciseren. Laat R een relatie zijn met attributen A1,A2,...,An. De verzameling attributen K = (Ai,Aj,...,Ak) van R

heet een kandidaatsleutel van R dan en slechts dan als de volgende twee, tijdsonafhankelijke eigenschappen gelden:

1. *Identificatie:*
   Nooit mogen twee verschillende tupels van R dezelfde waarde hebben èn voor Ai èn voor Aj,..., èn voor Ak.

2. *Minimaliteit:*
   Geen van de attributen Ai,Aj,...,Ak kan uit de verzameling K verwijderd worden zonder dat de uniciteitseigenschap verloren gaat.

Zoals reeds is uitgelegd, heeft iedere relatie tenminste één kandidaatsleutel. Neem ten gerieve van het voorbeeld even aan dat de attributen L.LNAAM en L.LNO beide de uniciteitseigenschap hebben (dat wil zeggen, dat nooit twee leveranciers dezelfde naam of hetzelfde nummer hebben). Dan heeft relatie L twee kandidaatsleutels, namelijk LNO en LNAAM. Kiezen we LNO als primaire sleutel (zoals in Fig. 4.2 gedaan is, getuige de PRIMARY KEY clausule), dan wordt LNAAM een alternatieve sleutel (en andersom).

Waarom zijn primaire sleutels zo belangrijk? Het antwoord is dat zij *in het relationele data model het enige adresseringsmechanisme op tupelniveau* aanbieden. Dat wil zeggen, dat de *enige* door het systeem gegarandeerde manier om een tupel precies te lokaliseren bestaat uit de combinatie (R,k), waarbij R de naam van de relatie en k de waarde van de primaire sleutel van het te lokaliseren tupel is. Dus de toegang via een waarde van een primaire sleutel levert (ten hoogste) een enkel tupel. Primaire sleutels worden op andere plaatsen in de database gebruikt om als referentie te dienen naar tupels die door de waarde van de primaire sleutel worden geïdentificeerd (zie ook de volgende paragraaf). Hieruit volgt dat primaire sleutels fundamenteel zijn voor het algehele functioneren van het relationele model en wel om precies dezelfde reden als waarom interne geheugenadressen fundamenteel zijn voor het functioneren van de onderliggende machine.

**4.4.2 Vreemde Sleutels.** Beschouw attribuut LNO van relatie LA van de leveranciers-artikelen database. Het is duidelijk, dat een gegeven waarde voor dat attribuut, zeg leveranciernummer L1, alleen maar in de relatie LA mag voorkomen als diezelfde waarde ook voorkomt als een waarde van de primaire sleutel LNO van de relatie L. Het zou bijvoorbeeld zinloos zijn als in relatie LA genoteerd staat dat een leverancier, zeg L1, een bepaald artikel kan leveren, terwijl deze leverancier L1 niet voorkomt in de relatie L. Op analoge wijze geldt dat een gegeven waarde voor attribuut ANO van relatie LA alleen maar mag voorkomen, als hij ook voorkomt als waarde voor attribuut ANO van relatie A. Het is onmogelijk een niet bestaand artikel te leveren.

Attributen LNO en ANO van relatie LA zijn voorbeelden van wat *vreemde sleutels* (foreign keys in Fig. 4.2) worden genoemd. In het

algemeen is een vreemde sleutel een attribuut (of attribuut combinatie) van een relatie R2, waarvan de waarden moeten overeenstemmen met waarden van de primaire sleutel van een relatie R1. Hierbij hoeven R1 en R2 niet verschillend te zijn. Merk op dat een vreemde sleutel en de corresponderende primaire sleutel gedefinieerd moeten zijn op hetzelfde onderliggende domein, anders is vergelijken niet zinvol.

In de relatie LA geldt dat LNO en ANO tezamen de primaire sleutel zijn. In het algemeen is het echter niet zo, dat een vreemde sleutel een deel moet zijn van de primaire sleutel van de relatie waar hij deel van uitmaakt. Voor een tegenvoorbeeld beschouwen we de volgende afdelingen-werknemers database:

```
AFD     ( AFDNO, ANAAM, MGR_WERKNO, BUDGET )
WERKN   ( WERKNO, WNAAM, AFDNO, SALARIS )
```

De primaire sleutels zijn respectievelijk afdelingnummer (AFDNO) en werknemernummer (WERKNO). Het attribuut WERKN.AFDNO is een vreemde sleutel in de relatie WERKN en is gerelateerd aan de primaire sleutel AFD.AFDNO van de relatie AFD; WERKN.AFDNO is echter geen deel van de primaire sleutel van relatie WERKN. Evenzo is MGR_WERKNO een vreemde sleutel in de relatie AFD (gerelateerd aan de primaire sleutel WERKNO van relatie WERKN) en ook deze vreemde sleutel is geen deel van de primaire sleutel.

Merk eveneens op, dat in de definitie van vreemde sleutel wordt gesteld dat R1 en R2 "niet noodzakelijk verschillend" hoeven te zijn. Hier volgt een voorbeeld van een relatie waarbij R1 en R2 gelijk zijn. Het betreft een uitbreiding van de relatie WERKN van hiervoor.

```
WERKN     ( WERKNO, WNAAM, AFDNO, SALARIS, MGR_WERKNO)
```

Opnieuw is WERKNO de primaire sleutel van de relatie WERKN. MGR_WERKNO (het werknemernummer van de manager van deze werknemer) is een vreemde sleutel in WERKN, die gerelateerd is aan de primaire sleutel van WERKN. Als bijvoorbeeld het tupel

```
( '111', 'Smit', '07', 50.000, '222')
```

in de relatie WERKN voorkomt, dan moet er ook een tupel met WERKNO = 222 in de relatie WERKN voorkomen (en dat is dan de manager van werknemer 111).

De gelijkheid "vreemde sleutel = primaire sleutel" stelt een referentie van de ene relatie naar de andere voor; dit soort referenties vormen de "lijm" die de database bij elkaar houdt. Een andere manier om dit te zeggen is dat dit soort gelijkheden betrekkingen tussen tupels representeren. Hierbij zij opgemerkt, dat niet alle betrekkingen tussen tupels door dit soort gelijkheden worden gerepresenteerd. Er is bijvoorbeeld een zekere betrekking tussen leveranciers en artikelen,

namelijk dat ze in dezelfde plaats gelokaliseerd zijn. Deze betrekking wordt voorgesteld door het WNPL attribuut van L en het PLAATS attribuut van A, maar deze attributen zijn *geen* vreemde sleutels, omdat WNPL en PLAATS geen primaire sleutels zijn.

## 4.5 DE TWEE INTEGRITEITSREGELS

We zijn nu zover dat we de twee integriteitsregels van het relationele model kunnen formuleren.

1. *Entiteit-integriteit:*
   Geen enkel attribuut dat deel uitmaakt van de primaire sleutel van een relatie mag een ongedefinieerde waarde aannemen.

2. *Referentie-integriteit:*
   Als de relatie R2 een vreemde sleutel VS bevat (die gerelateerd is aan de primaire sleutel PS van een relatie R1), dan moet elke waarde van VS in R2 òf
   (a) gelijk zijn aan de PS waarde in enig tupel van R1, òf
   (b) geheel ongedefinieerd zijn (dat wil zeggen dat elke attribuutwaarde die participeert in VS ongedefinieerd is).
   R1 en R2 hoeven niet verschillend te zijn.

Ongedefinieerde waarden duiden we aan als "null", hetgeen betekent "eigenschap niet van toepassing" of "waarde onbekend". Vanuit het standpunt van de twee integriteitsregels bekeken, betekent "null" een waarde die niet voorkomt in het domein waarop het attribuut is gedefinieerd (zo zou een negatief getal als null-waarde voor het domein HOEVHD kunnen fungeren).

De rechtvaardiging voor de entiteit-integriteitsregel is de volgende. Tupels van relaties corresponderen met entiteiten en associaties van het object systeem. (Zo correspondeert bijvoorbeeld de relatie L met de verzameling van leveranciers in het object systeem.) Per definitie, zie hoofdstuk 1, zijn entiteiten (en associaties) te onderscheiden; dat wil zeggen, ze bezitten op de een of andere manier een unieke identificatie. Primaire sleutels voeren in het relationele model de identificatie-functie uit. Een null waarde voor (een attribuut van) de primaire sleutel zou dan een innerlijke tegenspraak inhouden: feitelijk zou het betekenen dat er een entiteit bestaat die geen identiteit heeft (vandaar ook de naam van deze regel, namelijk de entiteit-integriteitsregel).

De rechtvaardiging voor de tweede regel ("referentie-integriteit") luidt als volgt: de belangrijkste bedoeling van deze regel is om te garanderen dat, als tupel t2 naar een zeker tupel t1 wijst, t1 dan ook bestaat. Het is daarom duidelijk dat bij een gegeven waarde (die niet null

is) voor de vreemde sleutel een zeker tupel in R1 moet horen, dat een gelijke waarde voor de primaire sleutel heeft. Soms kan het echter nodig zijn dat de vreemde sleutel null waarden aanneemt. Neem bijvoorbeeld aan dat het in een bepaald bedrijf is toegestaan dat een werknemer aan geen enkele afdeling is toegewezen. Het tupel dat deze werknemer representeert, zal dan voor het attribuut AFDNO de waarde null moeten bevatten.

## OPGAVEN

**4.1** In dit hoofdstuk is een groot aantal nieuwe termen ingevoerd. In vele gevallen zal het onderliggende begrip al wel tamelijk bekend zijn, maar in het algemeen hebben de relationele termen een nauwkeurigere betekenis dan de termen uit de traditionele gegevensverwerking. We hebben hieronder de nieuwe termen in twee kolommen gerangschikt; gevraagd wordt een definitie van deze termen te geven.

| | |
|---|---|
| relatie | graad |
| domein | cardinaliteit |
| attribuut | unaire relatie |
| tupel | binaire relatie |
| genormaliseerde relatie | null |
| primaire sleutel | enkelvoudig domein |
| relationele database | kandidaatsleutel |
| vreemde sleutel | alternatieve sleutel |
| extensie | intensie |

**4.2** Definieer een schema voor de oplossing van opgave 3.1 (de opgave over de personen-en-bekwaamheden database).

**4.3** Geef enige verschillen aan tussen de definitie van een relatie en een tabelweergave van een relatie. Neem daarbij Fig. 4.3 en de relatie ARTIKEL_STRUCT als voorbeeld.

**4.4** Geef een verschil aan tussen een algemene relatie en een relatie in 1NV.

**4.5** In een relationele database zijn onder andere de relaties R, S en T gedefinieerd. De primaire sleutel van R is het attribuut A, van S is het de combinatie van de twee vreemde sleutels B en C en van T de combinatie van de vreemde sleutels A en C. In T komt een tupel voor met a1 als waarde van attribuut A en c1 als waarde van attribuut C; in R een tupel met de waarde a2 voor het attribuut A. Bovendien is gegeven dat a2 niet in het A attribuut van T en c1 niet in het C attribuut van S voorkomt. Kan genoemde situatie zich voordoen en wat kunnen we zeggen over de inhoud van de relaties R, S en T?

**4.6** Auto's worden door een in verschillende landen opererende autofabriek A geleverd aan importeurs. Elk type wordt slechts in één kleur geleverd. Bij de importeur worden auto's eventueel van een andere kleur voorzien (al naargelang de landelijke behoefte). Per type kent men hier dus een aantal kleuren. De domeinen zijn TYPE = {josy, marty, justy} en KLEUR = {zwart, rood, wit, grijs}. Dit geeft bij een zekere importeur aanleiding tot de volgende relaties (kortheidshalve wordt de primaire sleutel onderstreept en de vreemde sleutel gecursiveerd):

A (TYPE,KLEUR, ...) met primaire sleutel TYPE en
I (*TYPE*,KLEUR, ...) met primaire sleutel (TYPE,KLEUR)
en vreemde sleutel TYPE.

Beantwoord de volgende vragen en motiveer het antwoord:

a. Hoeveel tupels kan de relatie A maximaal bevatten?
b. Hoeveel tupels kan de relatie I maximaal bevatten?
c. Kunnen de tupels (marty,zwart) en (marty,rood) gelijktijdig in relatie A voorkomen?
d. Kunnen de tupels (josy,zwart) en (josy,rood) gelijktijdig in relatie I voorkomen?
e. Kan het tupel (justy,rood) in de relatie A voorkomen als de relatie I uitsluitend het tupel (marty,rood) bevat?
f. Kan het tupel (justy,rood) in de relatie I voorkomen als de relatie A uitsluitend het tupel (marty,rood) bevat?

**4.7** Figuur 4.5 toont een extensie van een database die gegevens bevat betreffende leveranciers (L), artikelen (A) en projecten (P). Leveranciers, artikelen en projecten worden zoals gebruikelijk geïdentificeerd door leveranciernummer (LNO), artikelnummer (ANO) en projectnummer (PNO). De betekenis van een LAP tupel is dat de gespecificeerde leverancier het genoemde artikel levert aan het gespecificeerde project in de genoemde hoeveelheid (de combinatie LNO-ANO-PNO identificeert zo een tupel uniek). Definieer een schema voor deze database met gebruikmaking van de ingevoerde DDL.
N.B. Deze database zal nog in vele andere opgaven worden gebruikt.

L

| LNO | LNAAM | STATUS | WNPL |
|---|---|---|---|
| L1 | Smit | 20 | Londen |
| L2 | Janse | 10 | Parijs |
| L3 | Blaak | 30 | Parijs |
| L4 | Klerk | 20 | Londen |
| L5 | Adams | 30 | Athene |

A

| ANO | ANAAM | KLEUR | GEW | PLAATS |
|---|---|---|---|---|
| A1 | Moer | Rood | 12 | Londen |
| A2 | Bout | Groen | 17 | Parijs |
| A3 | Schroef | Blauw | 17 | Rome |
| A4 | Schroef | Rood | 14 | Londen |
| A5 | Nok | Blauw | 12 | Parijs |
| A6 | Kamrad | Rood | 19 | Londen |

P

| PNO | PNAAM | STAD |
|---|---|---|
| P1 | Scherm | Parijs |
| P2 | Ponser | Rome |
| P3 | Lezer | Athene |
| P4 | Console | Athene |
| P5 | Schijf | Londen |
| P6 | Terminal | Oslo |
| P7 | Band | Londen |

LAP

| LNO | ANO | PNO | HOEV |
|---|---|---|---|
| L1 | A1 | P1 | 200 |
| L1 | A1 | P4 | 700 |
| L2 | A3 | P1 | 400 |
| L2 | A3 | P2 | 200 |
| L2 | A3 | P3 | 200 |
| L2 | A3 | P4 | 500 |
| L2 | A3 | P5 | 600 |
| L2 | A3 | P6 | 400 |
| L2 | A3 | P7 | 800 |
| L2 | A5 | P2 | 100 |
| L3 | A3 | P1 | 200 |
| L3 | A4 | P2 | 500 |
| L4 | A6 | P3 | 300 |
| L4 | A6 | P7 | 300 |
| L5 | A2 | P2 | 200 |
| L5 | A2 | P4 | 100 |
| L5 | A5 | P5 | 500 |
| L5 | A5 | P7 | 100 |
| L5 | A6 | P2 | 200 |
| L5 | A1 | P4 | 100 |
| L5 | A3 | P4 | 200 |
| L5 | A4 | P4 | 800 |
| L5 | A5 | P4 | 400 |
| L5 | A6 | P4 | 500 |

**Fig. 4.5 De leveranciers-artikelen-projecten database.**

# 5
# Ontwerp van Relationele Databases I: Globaal

## 5.1 INLEIDING

Het doel van dit hoofdstuk is om een zeer globale (en dus onvolledige) schets te geven van een ontwerpmethode voor databases die moet resulteren in een relationeel schema. Hieruit volgt dat niet de eigenschappen, maar juist de gebruiksaspecten van het relationele data model aan de orde komen. In het vorige hoofdstuk werd reeds een aantal problemen met betrekking tot het bepalen van een relationeel schema geschetst. Door een bepaalde aanpak te volgen konden de benodigde relaties worden bepaald. De daar getoonde aanzet tot een ontwerpmethode voor relationele databases zal in dit hoofdstuk worden samengevat en verder uitgebreid. De grote lijn in die ontwerpmethode is het onderkennen en afbeelden op relaties van achtereenvolgens de entiteittypen, m:n associatietypen, 1:n associatietypen en kenmerken (in het bijzonder de niet-atomaire). (Deze termen werden in paragraaf 1.2 ingevoerd.) Tot de methode behoren ook het bepalen en specificeren van de integriteit van bijwerkoperaties (paragraaf 5.4). Een laatste onderdeel is normalisatie, een techniek die in het volgende hoofdstuk uitgebreid aan de orde komt. In paragraaf 5.6 wordt aangegeven wat normalisatie in essentie inhoudt en wat de rol ervan is bij het bepalen van een relationeel schema.

## 5.2 ENTITEITEN EN ASSOCIATIES

In deze paragraaf zullen we niet uitgaan van de leveranciers-artikelen database, omdat daarvoor het relationele schema al is gegeven. In

plaats daarvan beschouwen we het ontwerp van een relationeel schema van een bibliotheeksysteem.

Gegeven is een bibliotheek met boeken waarvan onder andere de titel, de auteur(s) en de inhoudsopgave moet worden bijgehouden. Als een boek niet één maar meer dan één auteur heeft, dan willen we de volgorde van deze auteurs weten (deze volgorde zegt namelijk vaak iets over de mate waarin de auteurs aan het boek hebben bijgedragen). Een inhoudsopgave bevat per hoofdstuk het nummer van het hoofdstuk, de titel en het nummer van de pagina waar het hoofdstuk begint. Een boek wordt uitgegeven door een uitgever en van een uitgever moeten naam, adres en type (wetenschappelijk, populair, etc.) worden vastgelegd. Verder worden van iedere auteur naam en adres bijgehouden.

**5.2.1 Entiteiten.** Een *entiteit* is ieder onderscheidbaar object waarover informatie moet worden bijgehouden. In het voorbeeld zijn dat de boeken, de uitgevers en de auteurs.

- *In het relationele data model worden entiteiten van hetzelfde type door relaties weergegeven.*

Zo zullen voor genoemde entiteittypen de relaties BOEKEN, UITGEVERS en AUTEURS worden ingevoerd. In het vorige hoofdstuk is al aangegeven dat de primaire sleutel tot taak heeft tupels (en dus entiteiten) te onderscheiden (de entiteit-integriteitsregel). Daarom moet de database-ontwerper voor iedere relatie in het ontwerp aangeven welk attribuut of welke attribuutcombinatie de primaire sleutel is.

De relaties voor de entiteiten worden in Fig. 5.1 gegeven (de domeinen zijn gemakshalve weggelaten).

**5.2.2 Associaties.** Associatietypen relateren op een een-op-een, veel-op-een, veel-op-veel (of veel-op-veel-op-veel, etc.) manier entiteiten van twee of meer typen. (Vanaf nu wordt het een-op-een associatietype als een speciaal geval van het veel-op-een associatietype beschouwd.) Van associatietypen hebben we in ons probleem eveneens een aantal voorbeelden, namelijk 'Auteur-Boek' (het veel-op-veel associatietype tussen Auteurs en Boeken: een auteur kan aan vele boeken meewerken en een boek kan door een aantal auteurs zijn geschreven), en 'Uitgave' (het een-op-veel associatietype tussen Uitgevers en Boeken: een boek wordt door één uitgever uitgegeven en een uitgever geeft vele boeken uit). Hoe worden associatietypen nu in het relationele model weergegeven?

```
RELATION    BOEKEN
    ( BOEKNO,
      PLAATSNUMMER,
      TITEL,
      INHOUDSOPGAVE ) , (* zie ook paragraaf 5.3 *)
    PRIMARY KEY   BOEKNO

RELATION     UITGEVERS
    ( UITG-NAAM,
      UITG-ADRES,
      TYPE ) ,
    PRIMARY KEY   UITG-NAAM

RELATION     AUTEURS
    ( AUT-NAAM,
      AUT-ADRES ) ,
    PRIMARY KEY   AUT-NAAM
```

**Fig. 5.1 Entiteiten van hetzelfde type in één relatie.**

- *De n:m associaties van hetzelfde type worden in het relationele data model door een aparte relatie weergegeven.*

In ons voorbeeld introduceren we de relatie AUTEUR-BOEK, waarvan elk tupel een auteur en een boek verbindt. Het blijkt dat een AUTEUR-BOEK relatie niet een 'natuurlijke' eigen primaire sleutel heeft. De vreemde sleutels zijn BOEKNO en AUT-NAAM en die vormen tezamen de primaire sleutel (ga dit na).

Om de volgorde van de auteurs van een boek aan te geven, introduceren we het attribuut VOLGNO. Een definitie van de relatie AUTEUR-BOEK ziet er als volgt uit.

```
RELATION    AUTEUR-BOEK
    ( BOEKNO,
      AUT-NAAM,
      VOLGNO ) ,
    PRIMARY KEY     (BOEKNO,AUT-NAAM) ,
    FOREIGN KEY     BOEKNO IDENTIFIES BOEKEN ,
    FOREIGN KEY     AUT-NAAM IDENTIFIES AUTEURS
```

**Fig. 5.2 Relatie met associaties van hetzelfde type: veel-op-veel geval.**

Veel-op-een associatietypen (n:1 associatietypen) kunnen op twee manieren gerepresenteerd worden.

De *eerste manier* is om n:1 associaties (van hetzelfde type) als aparte relatie te representeren, met als attributen de vreemde sleutels

naar de participerende entiteiten en de eventueel aanwezige eigen kenmerken. Zo kan het n:1 associatietype "uitgave", dat een uitgever en een boek verbindt door een ternaire relatie worden weergegeven met als attributen de vreemde sleutels naar boek en uitgever en het kenmerk "uitgavedatum". Dit zou resulteren in de volgende definitie:

```
RELATION    UITGAVE
    ( UITG-NAAM,
      BOEKNO,
      UITGV-DATUM ) ,
    PRIMARY KEY     (BOEKNO,UITG-NAAM) ,
    FOREIGN KEY     UITG-NAAM IDENTIFIES UITGEVERS ,
    FOREIGN KEY     BOEKNO IDENTIFIES BOEKEN
```

**Fig. 5.3 Relatie met associaties van hetzelfde type: veel-op-een geval.**

De *tweede manier* is om de veel-op-een associaties te *integreren* met de relatie die het 'veel' entiteittype weergeeft. Onder 'veel' entiteittype wordt bijvoorbeeld boek in het associatietype "uitgave" verstaan. Dit houdt in, dat de vreemde sleutel wordt beschouwd als een attribuut van het 'veel' entiteittype. Daarnaast kunnen de kenmerken van het n:1 associatietype worden gezien als kenmerken van het 'veel' entiteittype. In ons voorbeeld betekent dit dat het n:1 associatietype "uitgave" wordt opgenomen in de relatie BOEKEN. Deze relatie krijgt als vreemde sleutel een attribuut UITG-NAAM. Bovendien wordt de "uitgavedatum", die strikt genomen een kenmerk is van het n:1 associatietype "uitgave", opgenomen als attribuut van BOEKEN. Fig. 5.4 laat een en ander zien.

```
RELATION    BOEKEN
    ( BOEKNO,
      UITG-NAAM,
      UITG-DATUM,
      PLAATSNUMMER,
      TITEL,
      INHOUDSOPGAVE ) ,
    PRIMARY KEY     BOEKNO ,
    FOREIGN KEY     UITG-NAAM IDENTIFIES UITGEVERS
```

**Fig. 5.4 Associaties van hetzelfde type *geïntegreerd*: veel-op-een geval.**

*Evaluatie:* Welke van de twee zienswijzen verdient nu de voorkeur? De eerste zienswijze is consequenter: associaties worden beschouwd als een speciaal soort entiteiten en zouden dus in het relationele model als relaties moeten worden gerepresenteerd. Door n:1 associaties van hetzelfde type als relatie weer te geven, kan echter het aantal relaties sterk stijgen. Dit wordt als niet wenselijk ervaren, omdat de complexiteit

van het relationele schema hierdoor toeneemt en het formuleren van vragen ingewikkelder wordt. In het relationele data model worden daarom in het algemeen 1:n associatietypen *niet* als aparte relaties weergegeven. Deze benadering zal hier worden gevolgd, dus:

- *De veel-op-een associaties van hetzelfde type worden gerepresenteerd door attributen in de relatie van het 'veel' entiteittype. Kenmerken van zo'n associatie worden als attributen in dezelfde relatie opgenomen.*

## 5.3 NIET-ATOMAIRE KENMERKEN

Attributen moeten in het relationele model atomaire waarden aannemen, en kenmerken van entiteiten worden normaal gesproken door attributen weergegeven. Zodra kenmerken dus niet-atomaire waarden kennen, hebben we in het relationele data model een probleem. Een voorbeeld hiervan is het kenmerk "inhoudsopgave". Ieder boek heeft een inhoudsopgave met per hoofdstuk een hoofdstuknummer, titel en nummer van de eerste pagina. Het kenmerk inhoudsopgave is wat eerder een repeterende groep genoemd werd. De remedie is, zoals bekend, normaliseren (zie paragraaf 4.3). Een hoofdstuk *wordt binnen een boek* uniek door het hoofdstuknummer geïdentificeerd; we definiëren voor het niet-atomaire kenmerk een relatie INHOUDSOPGAVE. Voor deze relatie vormt het tweetal (HFDST-NO,BOEKNO) een primaire sleutel.

```
RELATION    INHOUDSOPGAVE
    ( BOEKNO,
      HFDST-NO,
      HFDST-TITEL,
      HFDST-PAGNO ) ,
    PRIMARY KEY    (HFDST-NO,BOEKNO) ,
    FOREIGN KEY     BOEKNO IDENTIFIES BOEKEN
```

**Fig. 5.5 Niet-atomaire kenmerken als relatie.**

Algemener:

*In het relationele data model worden niet-atomaire kenmerken door een aparte relatie weergegeven. Elke waarde die door een niet-atomair kenmerk wordt aangenomen, wordt een tupel in deze relatie.*

Opmerking.
Door het invoeren van de relatie INHOUDSOPGAVE kan en moet het attribuut INHOUDSOPGAVE van de relatie BOEKEN natuurlijk verdwijnen!

## 5.4 INTEGRITEIT EN BIJWERKOPERATIES

In het vorige hoofdstuk hebben we gezien dat twee integriteitsregels, de entiteit- en de referentie-integriteitsregel, deel uitmaken van het relationele model. Deze twee integriteitsregels hebben alleen betrekking op *toestanden* van de database. Elke (relationele) database-toestand die niet voldoet aan de twee regels, is per definitie incorrect. Een belangrijke vraag is nu hoe deze incorrecte toestanden *vermeden* kunnen worden. Hierover zeggen de integriteitsregels niets. In 5.4.1 en 5.4.2 wordt daar nader op ingegaan.

Een tweede probleem heeft te maken met de waarden die een vreemde sleutel mag aannemen. Staat het object systeem toe dat een vreemde sleutel ooit de waarde null krijgt, of niet? In het voorbeeld van de relatie INHOUDSOPGAVE kan men willen verbieden dat de vreemde sleutel BOEKNO de waarde null aanneemt. Null zou immers betekenen dat het onbekend is tot welk boek een bepaald hoofdstuk hoort. Dit lijkt niet reëel. In 5.4.3 en 5.4.4 wordt ingegaan op de restricties die er kunnen bestaan ten aanzien van het bijwerken van de waarde van een vreemde sleutel.

**5.4.1 Invoegen van tupel.** Het DBMS kan een operatie weigeren die, indien uitgevoerd, zou resulteren in een incorrecte database-toestand. Deze handelwijze kan het DBMS volgen om de *entiteit-integriteitsregel* te handhaven. Zodra een tupel (via een INSERT, zie paragraaf 7.6) voor invoegen wordt aangeboden, moet het DBMS controleren dat de primaire sleutel uniek en niet null is. Als deze controle negatief uitvalt, dan moet de INSERT worden geweigerd.

**5.4.2 Verwijderen van tupel.** Bij het handhaven van de *referentie-integriteitsregel* is het te prefereren dat het DBMS een bijwerkoperatie altijd accepteert en, als de integriteit in gevaar komt, compenserende operaties uitvoert. Het algehele resultaat moet een correcte database-toestand zijn. Als de gebruiker bijvoorbeeld vraagt auteur Blaak uit de relatie AUTEURS te verwijderen (via een DELETE, zie paragraaf 7.6), dan zou het DBMS, indien dat is gespecificeerd, ook zijn gehele oeuvre uit de database moeten verwijderen. Hiermee zou dan automatisch aan de referentie-integriteitsregel zijn voldaan. In een 'goed' DBMS moet het mogelijk zijn om te specificeren welke operaties geweigerd en welke geaccepteerd moeten worden. Bovendien, en dat is belangrijk, moet

voor de te accepteren operaties gespecificeerd worden welke compenserende operaties door het DBMS uitgevoerd moeten worden.

Om de referentie-integriteitsregel te handhaven, zijn de volgende drie DELETE opties mogelijk:

- CASCADES
  Het verwijderen van het AUTEURS tupel wordt door het DBMS gepropageerd. Dit houdt in dat ook de AUTEUR-BOEK tupels, die het oeuvre van deze auteur representeren, worden verwijderd.
- RESTRICTED
  Het tupel van AUTEURS wordt alleen verwijderd als er geen tupels van boeken van deze auteur meer in de database zijn opgenomen (anders wordt de operatie geweigerd; deze optie lijkt het meest zinnig in het onderhavige geval!).
- NULLIFIES
  De vreemde sleutel (AUT-NAAM) wordt in alle AUTEURS-BOEKEN tupels van deze auteur null gemaakt en daarna wordt het AUTEURS tupel verwijderd.

**5.4.3 Waarde van vreemde sleutel bij invoegen.** Welke waarden mag een vreemde sleutel aannemen als een tupel in de database wordt opgenomen?

Soms is het toegestaan dat een vreemde sleutel de waarde null (ongedefinieerd) heeft bij het opnemen van een tupel in de database. Stel dat een werknemer bij aanname in een bedrijf nog niet meteen aan een afdeling wordt toegewezen. 'Nog niet toegewezen' kan door de waarde null worden gerepresenteerd. In een pseudo-DDL wordt dit door ON INSERT NULL ALLOWED weergegeven.

Aan de andere kant is het moeilijk voorstelbaar dat er gegevens over een hoofdstuk in de database moeten worden opgenomen, terwijl het boek waarvan dit hoofdstuk een onderdeel is, nog niet bekend is. Dit zou in diezelfde pseudo DDL worden gespecificeerd door ON INSERT NULL NOT ALLOWED. Samenvattend krijgen we voor de INSERT-opdracht de volgende opties (tekst tussen vierkante haken is optioneel):

```
ON INSERT      NULL [ NOT ] ALLOWED
```

**5.4.4 Waarde van vreemde sleutel na wijzigen.** Stel dat een wijziging wordt aangebracht in de waarde van een vreemde sleutel. Welke waarden mag een vreemde sleutel, na wijziging (door een MODIFY opdracht, zie paragraaf 7.6) aannemen? Er zijn drie gevallen te onderscheiden, die aan de hand van een voorbeeld worden uitgelegd.

Stel dat een werknemer tot één, maar ook tot geen afdeling mag behoren. Stel bovendien dat hij van afdeling mag veranderen. In dat geval mag de vreemde sleutel AFDNO zonder restricties worden gewijzigd. Dit kan in de pseudo DDL worden weergegeven door

"MODIFY ALLOWED", hetgeen wil zeggen dat voor het wijzigen geen restricties bestaan.

Het kan ook zijn dat een vreemde sleutel, als hij eenmaal een niet-null waarde heeft aangenomen, altijd een niet-null waarde moet blijven houden. Stel dat in een bedrijf geldt dat een werknemer, na eenmaal aan een afdeling te zijn toegewezen, altijd tot een of andere afdeling moet behoren. In dat geval mag men AFDNO wel wijzigen, maar nooit NULL maken. Dit zou kunnen worden aangegeven door MODIFY ALLOWED EXCEPT TO NULL.

Het derde geval is het meest restrictief en betreft de situatie waarin een vreemde sleutel niet meer gewijzigd mag worden zodra een waarde ongelijk aan NULL is bereikt. Denk bijvoorbeeld aan het BOEKNO attribuut in een INHOUDSOPGAVE tupel. Als BOEKNO eenmaal een waarde heeft gekregen, dan mag deze waarde niet meer gewijzigd worden. In dat geval zou "MODIFY ALLOWED ONLY FROM NULL" moeten worden gespecificeerd. Voor de MODIFY krijgen we de volgende mogelijkheden. (Voor de betekenis van de metasymbolen, zie Appendix A.)

```
MODIFY  ALLOWED
        [ { EXCEPT TO NULL  |  ONLY FROM NULL } ]
```

Fig. 5.6 geeft een deel van het relationele schema van de bibliotheek database zoals het er nu uit ziet. In dit schema zijn specificaties ten aanzien van de semantiek van de bijwerkoperaties opgenomen. Er is gebruik gemaakt van de reeds ingevoerde pseudo DDL. Ga na wat voor de vreemde sleutel AUT-NAAM de INSERT en de MODIFY opties zijn.

Merk op dat de genoemde specificaties *alleen mogen afhangen van de situatie zoals die in het bedrijf gelden* en niet van de 'nukken' van een database-ontwerper. Opgemerkt dient te worden dat van hetgeen hiervoor is vermeld, nog maar weinig in de huidige relationele database systemen is gerealiseerd.

## 5.5 DE ONVOLLEDIGE ONTWERPMETHODE

Samenvattend zien de belangrijkste stappen van de ontwerpmethode voor een relationele database er als volgt uit. De methode wordt 'onvolledig' genoemd, omdat normalisatie er nog niet in is opgenomen.

1. Representeer entiteiten van hetzelfde type door een relatie en specificeer de primaire sleutel van die relatie.

```
RELATION    BOEKEN
      ( BOEKNO,
        UITG-NAAM,
        UITG-DATUM,
        PLAATSNUMMER,
        TITEL ) ,
      PRIMARY KEY    BOEKNO

RELATION    AUTEURS
      ( AUT-NAAM,
        AUT-ADRES ) ,
      PRIMARY KEY    AUT-NAAM

RELATION    AUTEUR-BOEK
      ( BOEKNO,
        AUT-NAAM,
        VOLGNO ) ,
      PRIMARY KEY  (AUT-NAAM,BOEKNO),
      FOREIGN KEY
          AUT-NAAM IDENTIFIES AUTEURS,
          DELETE OF AUTEURS RESTRICTED,
          ON INSERT NULL NOT ALLOWED,
          MODIFY  ALLOWED ONLY FROM NULL ,
      FOREIGN KEY
          BOEKNO IDENTIFIES BOEKEN,
          DELETE OF BOEKEN RESTRICTED,
          ON INSERT NULL NOT ALLOWED,
          MODIFY ALLOWED ONLY FROM NULL
```

**Fig. 5.6 Uitgebreide specificatie van bijwerkoperaties.**

2. Representeer ieder veel-op-veel associatietype als een relatie. Gebruik vreemde sleutels in die relatie om de participanten in de associatie aan te duiden. Specificeer de integriteitsregels die gelden ten aanzien van het bijwerken van de vreemde sleutels. Specificeer bovendien de sleutel van de relatie (waarschijnlijk dus de combinatie van de vreemde sleutels die de participanten identificeren).

3. Representeer elk niet-atomair kenmerk als een relatie. Specificeer de primaire sleutel van die relaties. Dit is waarschijnlijk de combinatie van de vreemde sleutel (dit is de primaire sleutel van de 'beschreven' entiteit, bijvoorbeeld BOEKNO) en de eigenschap die identificatie "binnen" de beschreven entiteit realiseert (bijvoorbeeld HFDST-NO).

4. Representeer elk veel-op-een associatietype als een vreemde sleutel in de relatie van het 'veel' entiteittype. Specificeer de DELETE restricties en de restricties die bij het bijwerken van deze vreemde sleutel horen. Neem de kenmerken van zo'n associatietype op als attributen van de genoemde relatie.

Door deze ontwerpmethode te volgen, krijgen we een ontwerp dat de volgende eigenschap heeft:

Iedere relatie bestaat uit:
- een primaire sleutel, waarmee de entiteiten van een bepaald type worden geïdentificeerd, tezamen met
- nul of meer additionele attributen, die additionele kenmerken representeren van het entiteittype dat door de primaire sleutel wordt geïdentificeerd (en niet van enig ander entiteittype).

In zo'n ontwerp bevat iedere relatie informatie over één en slechts één entiteittype. Het ontwerp zal eenvoudiger te begrijpen, eenvoudiger te gebruiken en gemakkelijker uit te breiden zijn dan een ontwerp waarin informatie over een aantal entiteittypen in een enkele relatie is samengebundeld.

## 5.6 NORMALISATIE

In deze paragraaf wordt enige aandacht aan de principes van normalisatie geschonken. (In het volgende hoofdstuk komen we uitvoerig op normalisatie terug.) Net als de zojuist beschreven methode, heeft normalisatie ten doel om aan een goed database-ontwerp bij te dragen. De vraag, die zich meteen al voordoet is: "Hoe is de samenhang tussen de ontwerpmethode van paragraaf 5.5 en normalisatie?". Het antwoord is dat de twee benaderingen *complementair* (aanvullend) zijn. Dat wil zeggen, dat in de praktijk de methoden tegelijk of na elkaar gebruikt zullen worden. Vaak wordt normalisatie afgeschilderd als het enige ontwerpmiddel dat nodig is; normalisatie is echter geen wondermiddel. Naar onze mening staat de ontwerpmethode zoals beschreven in 5.5, primair, waarbij normalisatie goed gebruikt kan worden om te kijken of er geen fouten zijn gemaakt bij het gebruik van genoemde ontwerpmethode. Na deze plaatsbepaling komen nu de basisprincipes van normalisatie aan de orde. Zoals al is opgemerkt wordt in het volgende hoofdstuk uitgebreider op dit onderwerp ingegaan.

In het relationele model voldoet iedere relatie aan de voorwaarde dat iedere attribuutwaarde atomair is. Zo'n relatie is in *eerste normaalvorm (1NV)* (of, anders gezegd, is genormaliseerd).

Het is mogelijk om verdere niveaus van normalisatie te definiëren: een tweede normaalvorm (2NV), derde normaalvorm (3NV), etc. Een

relatie is in 2NV als hij in 1NV is en bovendien nog aan een extra voorwaarde voldoet; hij is in 3NV als hij in 2NV is en bovendien een andere, extra voorwaarde vervult; etc. Dus elke normaalvorm is in zekere zin meer restrictief dan de vorige. Belangrijker is echter, dat iedere normaalvorm in het algemeen *wenselijker* is dan de vorige. Dit is zo, omdat de "(N+1)ste normaalvorm" bepaalde onaantrekkelijke eigenschappen *niet* heeft, die de "Nde normaalvorm" *wel* bezit. De argumentatie achter het stellen van de extra voorwaarde (die de overgang van de Nde naar de (N+1)ste NV mogelijk maakt) is nu juist om die onaantrekkelijke eigenschap kwijt te raken.

Beschouw bijvoorbeeld de relatie EERSTE, die in 1NV is, maar niet in 2NV. De primaire sleutel is (AUT-NAAM,BOEKNO).

EERSTE

| AUT-NAAM | BOEKNO | AUT-ADRES | VOLGNO |
|---|---|---|---|
| Blaak | B148 | Hof 3 | 1 |
| Blaak | B719 | Hof 3 | 1 |
| Adams | B719 | Plein 19 | 2 |
| Adams | B513 | Plein 19 | 1 |

De "onaantrekkelijke eigenschap" van EERSTE is duidelijk: het attribuut AUT-ADRES (dat het adres van een auteur geeft) bevat veel redundantie. Die redundantie zal leiden tot bijwerk- en consistentieproblemen die alleen kunnen worden vermeden door extra *controles* uit te voeren. Dat is ongewenst! Intuïtief staat AUT-ADRES op de verkeerde plek, omdat AUT-ADRES iets over een auteur zegt en niet over een boek dat door een auteur is geschreven. Een beter ontwerp is het volgende:

TWEEDE

| AUT-NAAM | BOEKNO | VOLGNO |
|---|---|---|
| Blaak | B148 | 1 |
| Blaak | B719 | 1 |
| Adams | B719 | 2 |
| Adams | B513 | 1 |

LW

| AUT-NAAM | AUT-ADRES |
|---|---|
| Blaak | Hof 3 |
| Adams | Plein 19 |

Deze relaties zijn in 2NV (in feite zijn ze ook in 3NV, etc.). Nu wordt slechts eenmaal bijgehouden, dat Blaak op Hof 3 en Adams op Plein 19 woont. Mocht Adams verhuizen, dan behoeven we slechts één tupel te wijzigen (geen gevaar voor inconsistentie).

Het globale doel van verdere normalisatie is redundanties als hiervoor geschetst weg te werken. De normalisatie-discipline voorziet in een verzameling richtlijnen, waarmee een relatie met bepaalde redundanties kan worden afgebroken in kleinere relaties die de redundanties niet bezitten. Het uiteindelijke doel is te eindigen met een

database-ontwerp waarin ieder feit slechts op één plaats voorkomt (zodat geen redundantie aanwezig is).

## 5.7 DE VOLLEDIGE ONTWERPMETHODE

Normalisatie houdt zich bezig met het reduceren van grotere relaties tot kleinere, waarbij successievelijk redundanties worden weggewerkt. Daarbij wordt er als het ware van uitgegaan dat al een (kleine) verzameling van grote relaties voorhanden is. Deze verzameling wordt zodanig bewerkt, dat een grotere verzameling kleinere relaties overblijft. Maar normalisatie zegt niets over de weg die leidt tot de verzameling relaties aan het begin. Anders gezegd, de vraag "Hoe komen we aan de kleine verzameling grote relaties?" wordt door normalisatie niet beantwoord, en dat is nu precies het thema waarmee de ontwerpmethode van paragraaf 5.5 zich bezighoudt. Daarom is in het begin van deze paragraaf ook beweerd dat de twee benaderingen elkaar aanvullen. De volledige ontwerpmethode ziet er dan als volgt uit:

1. Gebruik de ontwerpstappen van paragraaf 5.5 om relaties te genereren die entiteiten, associaties en niet-atomaire attributen representeren.

2. Gebruik daarna de normalisatie-discipline om te verifiëren of deze relaties zich aan de normalisatie-richtlijnen houden.

3. Herhaal het bovenstaande totdat het ontwerp compleet is.

## OPGAVEN

**5.1** Geef de integriteitsspecificatie bij de INHOUDSOPGAVE relatie van paragraaf 5.3.

**5.2** We beschouwen een bibliotheek database waarin artikelen en tijdschriften zijn opgenomen. Een artikel mag in verschillende tijdschriften verschijnen, bijvoorbeeld in de *Communications of the ACM* en in de *Transactions on Database Systems*. Daarnaast zijn er auteurs en onderwerpen. Iedere auteur kan vele artikelen schrijven en een artikel kan meer dan één auteur hebben. Bij een artikel hoort een lijst met kernwoorden die het artikel karakteriseren. Bovendien hoort een artikel bij slechts één onderwerp.
De entiteiten hebben de volgende kenmerken.

Onderwerp:
onderwerpnummer (uniek), naam van het onderwerp.
Artikel:
type (A = artikel, M = monografie), titel (uniek) en kernwoorden.
Tijdschrift:
type (M = maandblad, K = kwartaalblad), naam (uniek), uitgever.
Auteur:
naam (uniek), adres, instituut.

a. Geef de entiteittypen en de associatietypen aan. Per associatietype wordt gevraagd van welk soort ze zijn (1:n, etc.). Geef bovendien aan welke kenmerken niet atomair zijn.
b. Gevraagd wordt het relationele schema voor deze database te ontwerpen (hierbij mogen de domeinspecificaties worden weggelaten!). De attributen die de primaire sleutel vormen, moeten worden onderstreept met een doorlopende lijn; de vreemde sleutels met een stippellijn.

**5.3** Gegeven is de verkoopafdeling van een bedrijf. Klanten worden beschreven door naam (uniek) en adres. Klanten plaatsen orders, die bestaan uit een orderkop en enige orderregels. De orderkop bevat, naast een ordernummer (uniek), onder andere informatie over leverdatum en korting, terwijl een orderregel de hoeveelheid geeft die van een bepaald artikel besteld is. Artikelen hebben als kenmerken naam (uniek) en gewicht.
Beantwoord dezelfde vragen als in 5.2.

**5.4** Gegeven is een universiteit database waarin gegevens worden bijgehouden betreffende docenten, studenten en colleges. Van zowel de docenten als de studenten worden de naam (uniek), het adres en de woonplaats genoteerd. Verder zijn van een docent zijn kamernummer, van een student zijn leeftijd en van de colleges het nummer (uniek), de zaal en het tijdstip bekend. De database beschrijft ook dat elk college slechts door één docent gegeven wordt en dat elke student meer dan één college kan volgen.
Beantwoord dezelfde vragen als in 5.2.

# 6
# Ontwerp van Relationele Databases II: Normalisatie

## 6.1 INLEIDING

In het vorige hoofdstuk hebben we de plaats van normalisatie in de ontwerpmethode voor relationele databases gedefinieerd. Het begrip normaalvorm werd geïntroduceerd en het belang ervan uitgelegd. In dit hoofdstuk wordt nader op de techniek van het normaliseren ingegaan, waarbij nogmaals wordt benadrukt dat de database-ontwerper het verkrijgen van relaties in een hoge normaalvorm niet als wet moet zien, maar moet interpreteren als richtlijn. Soms zijn er goede redenen om de principes van het normaliseren in de wind te slaan. In paragraaf 6.6 geven we hier een voorbeeld van. De enige harde eis is, dat de relaties tenminste in eerste normaalvorm gebracht moeten worden. Database-ontwerp is voor 'grote databases' (dat wil zeggen, met veel relaties) een zeer complexe taak (voor 'kleine' is het meestal eenvoudig). Normalisatietechnieken kunnen bij deze taak behulpzaam zijn en komen daarom in dit hoofdstuk vrij uitgebreid aan de orde.

Hierbij zal weinig nadruk worden gelegd op een strakke, formele aanpak; veel meer wordt een beroep gedaan op intuïtie.

In paragraaf 6.2 wordt het eenvoudige, doch belangrijke begrip van 'functionele afhankelijkheid' geïntroduceerd. Dit begrip is de basis voor de oorspronkelijke drie normaalvormen voorgesteld door E.F. Codd [CODD70]). De twee daaropvolgende paragrafen behandelen dan de eerste (1NV), tweede (2NV) en derde normaalvorm (3NV). Bovendien komt de zogenaamde Boyce/Codd normaalvorm (BCNV) aan de orde. Paragraaf 6.5 gaat vervolgens in op het probleem van de alternatieve decompositie: het kiezen van de 'beste' decompositie van een relatie als

er een aantal equivalente BCNV decomposities zijn. Tenslotte volgt in de laatste paragraaf een samenvatting.

## 6.2 FUNCTIONELE AFHANKELIJKHEDEN

We beginnen onze behandeling van de normalisatietheorie met het definiëren van het fundamentele begrip *functionele afhankelijkheid.*

- Gegeven is een relatie R. Attribuut Y van R is *functioneel afhankelijk* van attribuut X van R, of in symbolen,

  R.X -> R.Y

  (lees "R.X bepaalt functioneel R.Y") dan en slechts dan als te allen tijde iedere X-waarde in R verbonden is met precies een Y-waarde in R. Hierbij mogen X en Y ook combinaties van attributen zijn.

In de leveranciers-artikelen database (zie voor het schema van deze database Fig. 4.2) zijn de attributen LNAAM, STATUS en WNPL functioneel afhankelijk van het attribuut LNO van relatie L, omdat er voor een zekere waarde van L.LNO precies één waarde voor elk van de attributen L.LNAAM, L.STATUS en L.WNPL bestaat. Dit geldt niet alleen voor de in dit boek gegeven database-inhouden, maar voor alle database-inhouden die zouden kunnen optreden. Waarom kunnen we dit zo stellig beweren? Omdat genoemde functionele afhankelijkheden in het object systeem wortelen. Dit object systeem vertelt ons dat een leverancier een uniek leveranciernummer heeft, dat elke leverancier slechts één en niet meer dan één naam, status en woonplaats heeft. Dan moet dus noodzakelijkerwijs de relatie, die informatie over leveranciers representeert, dit weerspiegelen. Dus zal te allen tijde bij een gegeven LNO-waarde slechts één WNPL-waarde horen, etc. In symbolen wordt dit als volgt weergegeven.

L.LNO -> L.LNAAM
L.LNO -> L.STATUS
L.LNO -> L.WNPL

of, nog korter genoteerd:

L.LNO -> L.(LNAAM,STATUS,WNPL)

Merk op dat alle attributen Y van relatie R zonder meer functioneel afhankelijk zijn van attribuut X als X een kandidaatsleutel van relatie R is (dit volgt uit de definitie van kandidaatsleutel). Dit geldt dus ook als X de primaire sleutel is. De zojuist gegeven voorbeelden met betrekking tot relatie L illustreren dit punt. Evenzo geldt voor de relatie LA:

LA.(LNO,ANO) -> LA.HOEV

Functionele afhankelijkheid zullen we vanaf nu aanduiden met "FA" (met "FA'en" als meervoud). Merk op dat de definitie van functionele afhankelijkheid niet eist dat X een kandidaatsleutel moet zijn. Met andere woorden, er is geen eis dat een gegeven X-waarde slechts eenmaal in een tupel van R mag voorkomen. Er kunnen dus ook FA'en zijn waarin X niet een kandidaatsleutel is. We geven een alternatieve definitie van FA, die dit punt wat explicieter stelt.

- Gegeven een relatie R. Attribuut Y van R is *functioneel afhankelijk* van attribuut X van R dan en slechts dan als te allen tijde voor elk tweetal tupels t1 en t2 van R uit de gelijkheid van de X-waarde in t1 en t2 de gelijkheid van de Y-waarde volgt.

Bijvoorbeeld, de relatie EERSTE van paragraaf 5.6 vervult de FA:

EERSTE.AUT-NAAM -> EERSTE.AUT-ADRES

(elk EERSTE tupel met een gegeven AUT-NAAM-waarde moet dezelfde AUT-ADRES-waarde hebben). Toch is AUT-NAAM geen kandidaatsleutel.

We definiëren ook het begrip volledig functioneel afhankelijk.

- Attribuut Y van relatie R is *volledig functioneel afhankelijk* van de attribuutcombinatie X van relatie R als het functioneel afhankelijk is van X en niet functioneel afhankelijk is van een echte deelverzameling van X (dat wil zeggen, dat er geen echte deelverzameling Z van X bestaat, waarvoor geldt dat Y functioneel afhankelijk is van Z).

Bijvoorbeeld, in relatie L geldt zeker dat attribuut WNPL functioneel afhangt van de combinatie (LNO,STATUS):

L.(LNO,STATUS) -> L.WNPL

Dit is echter geen volledige afhankelijkheid, omdat de volgende FA ook geldt:

L.LNO -> L.WNPL

(WNPL is ook al functioneel afhankelijk van LNO alleen). Merk op dat als Y wel functioneel, doch niet volledig functioneel afhangt van X, dat dan X een combinatie van attributen is.

Het is gemakkelijk de FA'en in een gegeven relatie door middel van een *functioneel afhankelijkheidsdiagram* (FA diagram) weer te geven. Het FA diagram voor de relaties L, A en LA wordt gegeven in Fig. 6.1.

We benadrukken nog eens dat functionele afhankelijkheid een semantisch begrip is. Het opsporen van de FA'en is een deel van het proces om de betekenis van gegevens te bepalen. Het feit dat WNPL functioneel afhankelijk is van LNO betekent bijvoorbeeld dat er een restrictie in het object systeem is die zegt, dat iedere leverancier in precies één woonplaats gelokaliseerd is. Omdat zij een onderdeel van de semantiek van het gegeven probleem is, moet die restrictie op de een of andere wijze door de database gehonoreerd worden. De manier om dit af te dwingen is door de restrictie in het relationele schema te specificeren. We zullen zien dat normalisatie leidt tot een zeer eenvoudige manier van FA specificatie.

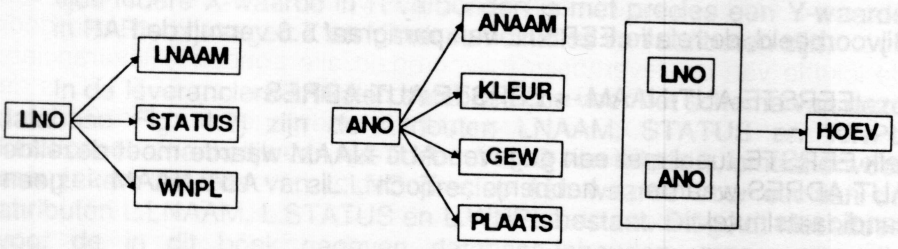



**Fig. 6.1 Functionele afhankelijkheden in de relaties L,A, LA.**

## 6.3 EERSTE, TWEEDE EN DERDE NORMAALVORMEN

We beginnen met het uitleggen van de termen 'niet-sleutel attribuut' en 'wederzijds onafhankelijk'. In deze paragraaf nemen we aan dat elke relatie een primaire sleutel heeft en *geen* alternatieve sleutels! Het geval van meer dan één kandidaatsleutel wordt in paragraaf 6.4 behandeld.

- Een *niet-sleutel attribuut* van relatie R is een attribuut dat niet participeert in de primaire sleutel van R.
- Twee (of meer) attributen zijn wederzijds onafhankelijk als geen van de betrokken attributen functioneel afhankelijk is van (een van) de andere. Merk op dat zulk soort attributen onafhankelijk van elkaar kunnen worden bijgewerkt: er behoeven geen afhankelijkheden in acht te worden genomen.

We beginnen met het herhalen van de definitie van de eerste normaalvorm.

- Een relatie R is in eerste normaalvorm (1NV) dan en slechts dan als alle onderliggende domeinen uitsluitend atomaire waarden bevatten.

Een relatie die alleen in 1NV is en niet in 2NV (en dus niet in 3NV, etc.) heeft een structuur die om een aantal redenen ongewenst is. Om dit punt te illustreren voeren we opnieuw relatie EERSTE in. Deze relatie bevat informatie over leveranciers en leverbare artikelen. Deze informatie is dus niet zoals gebruikelijk gesplitst over twee relaties L en LA.

EERSTE ( LNO, STATUS, WNPL, ANO, HOEV )

De attributen hebben de gebruikelijke betekenis. Om het voorbeeld echter interessant te maken voor de normalisatie-technieken nemen we aan dat STATUS functioneel afhankelijk is van WNPL. De betekenis van deze restrictie is, dat de status van een leverancier wordt bepaald door de lokatie van die leverancier (bijvoorbeeld alle Londense leveranciers *moeten* een status 20 hebben). De primaire sleutel van EERSTE is de combinatie (LNO,ANO). Het FA diagram is weergegeven in Fig. 6.2. Wat opvalt is dat in het FA diagram naast de pijlen die vanuit de primaire sleutel lopen, ook nog extra pijlen aanwezig zijn. Het zal blijken dat deze extra pijlen problemen veroorzaken!

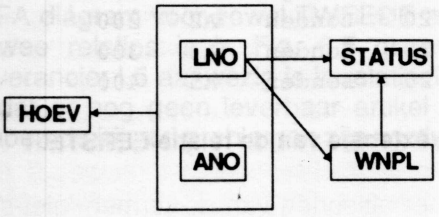



**Fig. 6.2 Functionele afhankelijkheden in de relatie EERSTE.**

Fig. 6.3 laat een extensie van relatie EERSTE in tabelvorm zien. Men kan nagaan, dat de functionele afhankelijkheden als geschetst in het FA diagram van Fig. 6.2 in acht worden genomen. De redundanties zijn duidelijk. Ieder tupel voor leverancier L1 representeert het feit dat L1 in Londen woont; evenzo toont ieder tupel waarin Londen als WNPL voorkomt dat de status 20 is.

Met betrekking tot de relatie EERSTE gelden zogenaamde bijwerkanomalieën: moeilijkheden met de drie bijwerkoperaties INVOEGEN, VERWIJDEREN en WIJZIGEN. Laten we ons eens

concentreren op de FA EERSTE.LNO -> EERSTE.WNPL. Problemen treden op bij alle drie bijwerkoperaties.

**INVOEGEN**

We kunnen het feit dat een bepaalde leverancier een bepaalde woonplaats als lokatie heeft, niet noteren totdat deze leverancier een artikel kan leveren. Zo geeft Fig. 6.3 niet aan dat leverancier L5 in Athene zit. De reden is dat, zolang L5 geen artikel kan leveren, we geen adequate sleutelwaarde hebben. (Bedenk dat de entiteit-integriteitsregel eist dat geen enkel attribuut van de primaire sleutel null (onbekend) mag zijn; in relatie EERSTE is de primaire sleutel de combinatie (LNO,ANO).)

| EERSTE | LNO | STATUS | WNPL | ANO | HOEV |
|---|---|---|---|---|---|
| | L1 | 20 | Londen | A1 | 300 |
| | L1 | 20 | Londen | A2 | 200 |
| | L1 | 20 | Londen | A3 | 400 |
| | L1 | 20 | Londen | A4 | 200 |
| | L1 | 20 | Londen | A5 | 100 |
| | L1 | 20 | Londen | A6 | 100 |
| | L2 | 10 | Parijs | A1 | 300 |
| | L2 | 10 | Parijs | A2 | 400 |
| | L3 | 10 | Parijs | A2 | 200 |
| | L4 | 20 | Londen | A2 | 200 |
| | L4 | 20 | Londen | A4 | 300 |
| | L4 | 20 | Londen | A5 | 400 |

**Fig. 6.3 Een extensie van de relatie EERSTE.**

**VERWIJDEREN**

Als we het enige EERSTE tupel van een leverancier verwijderen, dan verwijderen we niet alleen de informatie betreffende een leverbaar artikel, maar ook de informatie dat die leverancier in een bepaalde plaats gelokaliseerd is. Als we bijvoorbeeld het EERSTE tupel met LNO = L3 en ANO = A2 verwijderen, dan kunnen we niet meer achterhalen dat L3 in Parijs woont.

**WIJZIGEN**

De woonplaats van een leverancier kan vele malen in EERSTE verschijnen. Deze redundantie veroorzaakt bijwerkproblemen. Als bijvoorbeeld L1 van Londen naar Amsterdam verhuist, dan moet de gebruiker (en niet het database systeem, dat van die redundantie niets weet) EERSTE helemaal doorzoeken en ieder L1 tupel bijwerken

(overigens kan dat, afhankelijk van het niveau waarop geprogrammeerd wordt, soms met een enkele opdracht afgedwongen worden). Bovendien bestaat het gevaar van inconsistentie: er hoeft maar één programma te zijn dat het bijwerken niet goed verzorgt en inconsistentie kan een feit zijn (dit laatste wil zeggen dat volgens sommige tupels Amsterdam de woonplaats van L1 is en volgens andere Londen).

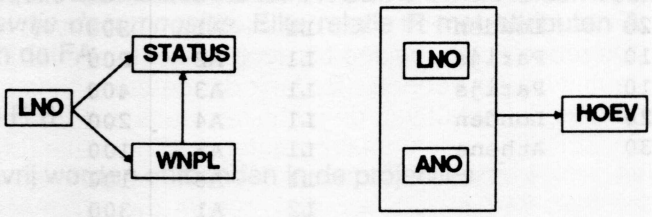



**Fig. 6.4 Functionele afhankelijkheden in de relaties TWEEDE en LA.**

De oplossing voor genoemde problemen is om de relatie EERSTE te vervangen (te reduceren) tot twee relaties, ieder met zijn 'eigen' FA'en. In dit geval worden dat de relaties

TWEEDE ( LNO, STATUS, WNPL )

en

LA ( LNO, ANO, HOEV )

In Fig. 6.4 is het FA diagram voor zowel TWEEDE als LA getekend. De inhoud van de twee relaties is in Fig. 6.5 gegeven. Merk op, dat informatie over leverancier L5 aanwezig is in relatie TWEEDE, maar niet in relatie LA (omdat er nog geen leverbaar artikel voor L5 is). Het is duidelijk dat voornoemde bijwerkproblemen zijn verdwenen:

**INVOEGEN**

We kunnen de informatie dat L5 in Athene woont in de relatie TWEEDE opnemen, ook al heeft L5 nog geen leverbaar artikel. (In feite geeft Fig. 6.5 die situatie al weer.)

**VERWIJDEREN**

We kunnen het feit dat artikel A2 door leverancier L3 leverbaar is, verwijderen door het overeenkomstige tupel van LA te verwijderen. De informatie dat L3 in Parijs woont, is in relatie TWEEDE opgeslagen.

**WIJZIGEN**

In de gereviseerde structuur (zie Fig. 6.5) komt de woonplaats van een leverancier slechts eenmaal voor. De LNO-WNPL redundantie is

verwijderd. Zo kunnen we de woonplaats van L1 wijzigen van Londen naar Amsterdam door het relevante tupel (niet "de tupels") van TWEEDE aan te passen.

TWEEDE

| LNO | STATUS | WNPL |
|---|---|---|
| L1 | 20 | Londen |
| L2 | 10 | Parijs |
| L3 | 10 | Parijs |
| L4 | 20 | Londen |
| L5 | 30 | Athene |

LA

| LNO | ANO | HOEV |
|---|---|---|
| L1 | A1 | 300 |
| L1 | A2 | 200 |
| L1 | A3 | 400 |
| L1 | A4 | 200 |
| L1 | A5 | 100 |
| L1 | A6 | 100 |
| L2 | A1 | 300 |
| L2 | A2 | 400 |
| L3 | A2 | 200 |
| L4 | A2 | 200 |
| L4 | A4 | 300 |
| L4 | A5 | 400 |

**Fig. 6.5 De inhoud van TWEEDE en LA.**

Als we Fig. 6.4 vergelijken met Fig. 6.2 dan zien we dat door het effect van de structuuraanpassing de niet-volledige afhankelijkheden zijn geëlimineerd. Deze eliminatie blijkt de problemen te hebben opgelost. In relatie EERSTE beschrijft het attribuut WNPL niet de entiteit die door de primaire sleutel wordt geïdentificeerd, namelijk een leverbaar artikel, maar de leverancier die het artikel kan leveren. Op deze wijze bereiken we dat een kenmerk door een attribuut in de relatie van *dat* entiteittype wordt gerepresenteerd, dat het meest direct door het kenmerk wordt beschreven. Dit gold in de relatie EERSTE niet voor WNPL en STATUS.

We geven nu een definitie van de tweede normaalvorm.

- Een relatie R is in *tweede normaalvorm (2NV)* dan en slechts dan als de relatie in 1NV is en elk niet-sleutel attribuut volledig van de primaire sleutel afhangt.

De relaties TWEEDE en LA zijn beide in 2NV (de primaire sleutels zijn respectievelijk LNO en de combinatie (LNO,ANO)). Relatie EERSTE is niet in 2NV. Een relatie die in 1NV is en niet in 2NV, kan altijd tot een equivalente verzameling relaties in 2NV worden gereduceerd. Merk op dat een relatie die in 1NV en niet in 2NV is, als primaire sleutel een combinatie van attributen moet hebben. Het reductieproces bestaat uit het vervangen van de 1NV relatie door geschikte *projecties*. De verzameling projecties die zo verkregen wordt, is equivalent aan de oorspronkelijke relatie in de zin dat de oorspronkelijk relatie altijd weer

verkregen kan worden door de *natuurlijke join* van die projecties te nemen. Dat wil zeggen, er is geen informatie verloren gegaan door het reduceren (hetgeen natuurlijk belangrijk is), en het reductieproces is omkeerbaar. In ons voorbeeld zijn TWEEDE en LA de projecties van EERSTE en EERSTE is de natuurlijke join van TWEEDE en LA over LNO.

De reductie van EERSTE tot TWEEDE en LA is een voorbeeld van een *verliesvrije decompositie*. Elke relatie R met attributen A, B, C, die voldoet aan de FA

R.A -> R.B

kan verliesvrij worden ontbonden in de projecties

R1(A,B) en R2(A,C).

Dit resultaat is voor het eerst bewezen door Heath en wordt soms de stelling van Heath genoemd [HEAT71]. Omdat er geen informatie in het reductieproces verloren gaat, kan elke informatie die van de oorspronkelijke structuur kan worden afgeleid ook van de nieuwe structuur worden afgeleid. Het omgekeerde is niet waar, immers: de nieuwe structuur kan informatie bevatten (zoals het feit dat L5 in Athene woont) die niet kan worden gerepresenteerd in de oorspronkelijke structuur. In die zin kan de nieuwe structuur worden gezien als een wat meer betrouwbaarder representatie van het object systeem.

De TWEEDE/LA structuur veroorzaakt echter nog steeds problemen. Met de relatie LA zijn we tevreden (LA zal daarom verder genegeerd worden). Relatie TWEEDE echter, heeft nog steeds te 'weinig' wederzijdse onafhankelijkheid bij haar niet-sleutel attributen. Het FA diagram voor TWEEDE (Fig. 6.4) laat zien dat STATUS weliswaar volledig afhankelijk is van LNO, maar dat deze afhankelijkheid (ook) *transitief* (via WNPL) is: iedere LNO-waarde bepaalt een WNPL-waarde en die WNPL-waarde bepaalt op haar beurt weer de STATUS-waarde. In het algemeen geldt, dat als de FA'en

R.A -> R.B en R.B -> R.C

beide gelden, dan geldt ook de transitieve FA

R.A -> R.C

Transitieve afhankelijkheden leiden tot bijwerkanomalieën. Om dit aan te tonen concentreren we ons nu op de FA TWEEDE.WNPL -> TWEEDE.STATUS (zie Fig. 6.5), die onder andere de WNPL/STATUS redundantie in TWEEDE veroorzaakt.

**INVOEGEN**

We kunnen de informatie dat een bepaalde woonplaats een bepaalde status heeft niet kwijt in de database zoals hij er nu uit ziet. We kunnen bijvoorbeeld niet stellen dat leveranciers die in Rome gevestigd zijn een status van 50 hebben, totdat we echt een leverancier uit Rome hebben. De reden is dat we tot het moment dat een leverancier zich meldt, geen geschikte primaire sleutelwaarde hebben (primaire sleutel is LNO).

**VERWIJDEREN**

Als we het enige TWEEDE tupel van een bepaalde woonplaats verwijderen, dan vernietigen we niet alleen de informatie over de betreffende leverancier, maar ook de informatie dat die woonplaats een bepaalde status heeft. Bijvoorbeeld, als we het TWEEDE tupel voor L5 verwijderen, zijn we de informatie dat Athene een status 30 heeft ook kwijt. (Er zij nog eens op gewezen dat de invoeg- en verwijderanomalieën twee aspecten van hetzelfde probleem zijn.)

**WIJZIGEN**

De status van een gegeven woonplaats komt in het algemeen vele malen in TWEEDE voor (de relatie bevat nog steeds redundantie). Als we de status van Londen veranderen van 20 in 30, dan moeten we TWEEDE helemaal doorlopen en alle "Londense tupels" bijwerken. Zoals bekend dreigt bij redundantie het gevaar van inconsistentie.

De oplossing voor genoemde problemen bestaat er weer uit dat we de oorspronkelijk relatie (TWEEDE in dit geval) vervangen door twee projecties, namelijk de projecties

LW ( LNO, WNPL )

en

WS ( WNPL, STATUS ).

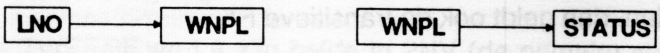



**Fig. 6.6 Functionele afhankelijkheden in de relatie LW en WS.**

Het FA diagram voor deze twee relaties is in Fig.6.6 getekend; een voorbeeld van een mogelijke extensie van de relaties LW en WS wordt in Fig. 6.7 in tabelvorm gegeven. Merk op dat de statusinformatie van Rome, zijnde 50, in de relatie WS is opgenomen. Nogmaals, het

reductieproces is omkeerbaar, omdat TWEEDE de natuurlijke join van LW en WS over WNPL is.

LW

| LNO | WNPL |
|---|---|
| L1 | Londen |
| L2 | Parijs |
| L3 | Parijs |
| L4 | Londen |
| L5 | Athene |

WS

| WNPL | STATUS |
|---|---|
| Athene | 30 |
| Londen | 20 |
| Parijs | 10 |
| Rome | 50 |

**Fig. 6.7. Extensie van LW en WS in tabelvorm.**

Deze gereviseerde structuur lost alle eerder genoemde bijwerkproblemen op. Als we Fig. 6.6 en Fig. 6.4 vergelijken, dan zien we dat het effect van de verdere structurering is dat de transitieve afhankelijkheid van STATUS ten opzichte van LNO is weggewerkt. Hierdoor zijn de moeilijkheden verdwenen. In de relatie TWEEDE worden leveranciers beschreven (de primaire sleutel van TWEEDE is LNO). Het attribuut STATUS hoort echter bij de entiteit woonplaats en niet bij de entiteit leverancier. STATUS beschrijft een woonplaats waar natuurlijk leveranciers gelokaliseerd kunnen zijn, dat wel. Het vermengen van deze twee soorten informatie in dezelfde relatie veroorzaakt de problemen. Het attribuut WNPL in de relatie LW is een vreemde en WNPL in WS zelfs een primaire sleutel!

We geven nu een definitie van de derde normaalvorm.

- Een relatie R is in *derde normaalvorm (3NV)* dan en slechts dan als de relatie in 2NV is en ieder niet-sleutel attribuut niet transitief afhankelijk is van de primaire sleutel.

De relaties LW en WS zijn beide in 3NV (de primaire sleutels zijn LNO en WNPL). Relatie TWEEDE is niet in 3NV. Een relatie die in 2NV en niet in 3NV is, kan altijd tot een equivalente verzameling relaties in 3NV worden herleid. We hebben reeds aangegeven dat dit proces omkeerbaar is en dat dus geen informatie in dit proces verloren gaat. Echter, de verzameling relaties in 3NV kan informatie bevatten, zoals het feit dat Rome een status van 50 heeft, die niet in de oorspronkelijke 2NV relatie te representeren is. Op dezelfde wijze als de TWEEDE/LA structuur een iets betere manier van representeren van het object systeem geeft dan de 1NV relatie EERSTE, zo is de LW/WS structuur een iets betere structuur dan de 2NV relatie TWEEDE.

We besluiten deze paragraaf door te benadrukken dat normalisatie een kwestie is die met de interpretatie van de gegevensstructuur te

maken heeft en niet met de waarden die toevallig op een zeker moment in een relatie voorkomen. Het is *niet* mogelijk om naar de extensie van een zekere relatie te kijken en dan te bepalen of de relatie in 3NV is of niet; het is nodig om de betekenis (semantiek) van de attributen te kennen voordat een oordeel kan worden geveld. In het bijzonder geldt dat het DBMS een relatie niet in, bijvoorbeeld, 3NV kan houden zonder geïnformeerd te zijn over alle relevante afhankelijkheden. Voor een relatie in 3NV is de specificatie van de primaire sleutel alles wat nodig is om het DBMS adequaat te informeren. Voor een relatie die niet in 3NV is, zijn er extra specificaties nodig (zie bijvoorbeeld Fig. 6.4, WNPL -> STATUS).

Merk op dat het DBMS kan controleren of een relatie in 3NV op een zeker moment een adequate inhoud heeft door na te gaan of er geen duplicaat primaire sleutelwaarden zijn.

## 6.4 BOYCE/CODD NORMAALVORM

Tot nu toe hebben we steeds relaties beschouwd die slechts *één* kandidaatsleutel hebben. Eventueel mag deze kandidaatsleutel samengesteld zijn, dat wil zeggen, bestaan uit een combinatie van attributen (zie bijvoorbeeld de relatie LA). Op deze situatie is Codd's definitie van 3NV gebaseerd. Het blijkt echter dat problemen *kunnen* optreden zodra een relatie

1. meer dan één kandidaatsleutel heeft,

2. die kandidaatsleutels samengesteld zijn, en

3. de kandidaatsleutels overlappend zijn, dat wil zeggen tenminste één attribuut gemeenschappelijk hebben.

In deze paragraaf gaan we eerst een relatie bespreken die meer dan één kandidaatsleutel heeft, maar waarbij de sleutels uit slechts één attribuut bestaan, zodat de sleutels niet overlappend zijn. Het zal blijken dat hier geen problemen optreden. Daarna bespreken we een relatie met twee, overlappende kandidaatsleutels, en hier treedt dan een probleem op. De oplossing van het probleem is weer gelegen in het reduceren (via projecties) van de relatie tot twee relaties. Deze relaties zijn in de zogenaamde Boyce/Codd normaalvorm (BCNV). Van deze situatie worden twee voorbeelden gegeven. Het derde en laatste geval dat we bespreken handelt ook over een relatie met overlappende kandidaatsleutels, echter hier treden geen problemen op. Dit voorbeeld illustreert dat er niet altijd problemen optreden als er sprake is van overlappende kandidaatsleutels.

We geven eerst de definitie van BCNV. Daarvoor hebben we het begrip determinant nodig.

- Een *determinant* van een relatie R is een attribuut (of attribuutcombinatie) van R waarvan tenminste één ander attribuut volledig functioneel afhankelijk is.

In de relatie EERSTE zijn bijvoorbeeld de attributen LNO, WNPL en (LNO,ANO) determinanten. BCNV is nu als volgt gedefinieerd.

- Een relatie R is in Boyce/Codd normaalvorm (BCNV) dan en slechts dan als iedere determinant een kandidaatsleutel is.

Merk op dat BCNV begripsmatig eenvoudiger is dan 3NV, omdat er geen verwijzing is naar andere normaalvormen (2NV en daarin weer naar 1NV) en omdat het begrip transitieve afhankelijkheid ontbreekt. Ofschoon BCNV een sterkere eis aan een relatie stelt dan 3NV (iedere BCNV relatie is in 3NV, maar er zijn relaties die in 3NV zijn maar niet in BCNV) blijkt het mogelijk te zijn elke relatie door een verliesvrije decompositie in een equivalente verzameling BCNV relaties te brengen.

Laten we eens kijken of de relaties EERSTE en TWEEDE die niet in 3NV zijn misschien wel in BCNV zijn. Relatie EERSTE bevat drie determinanten, namelijk LNO, WNPL en (LNO,ANO). Van deze drie is alleen (LNO,ANO) een kandidaatsleutel, dus EERSTE is niet in BCNV. Evenzo is TWEEDE niet in BCNV, omdat de determinant WNPL niet een kandidaatsleutel is. Aan de andere kant zijn de relaties LA, LW en WS alle in BCNV, omdat in die gevallen de primaire sleutel de enige determinant in de relatie is.

We gaan nu de drie hiervoor genoemde gevallen bekijken.

**6.4.1 Twee enkelvoudige kandidaat sleutels**. Neem aan dat in de leveranciers relatie L zowel LNO als LNAAM kandidaatsleutel zijn, dat wil zeggen dat iedere leverancier naast een uniek leveranciernummer ook nog een unieke naam heeft. Vanzelfsprekend zijn die sleutels *niet overlappend*, want ze zijn enkelvoudig.

L ( LNO, LNAAM, STATUS, WNPL )

Neem verder aan, zoals eerder in dit boek is gedaan, dat WNPL en STATUS wederzijds functioneel onafhankelijk zijn. Dan wordt het FA diagram voor L gegeven door Fig. 6.8.

Relatie L is in BCNV. Ofschoon het FA diagram "complexer" is dan het FA diagram van een relatie in 3NV, is het toch zo dat de enige determinanten kandidaatsleutels zijn; dat wil zeggen dat de enige pijlen die in Fig. 6.8 getekend zijn, vertrekken vanuit LNO of LNAAM. Merk echter op dat het wenselijk is in het relationele schema het feit te

specificeren dat LNAAM een kandidaatsleutel is, zodat het DBMS de vereiste uniciteit kan afdwingen. Een hypothetische syntax is de volgende:

```
RELATION  L   ( LNO, LNAAM, STATUS, WNPL ) ,
PRIMARY KEY    LNO ,
ALTERNATE KEY  LNAAM
```

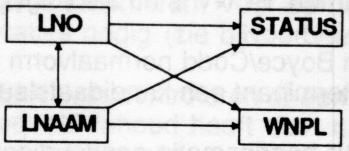



**Fig. 6.8 Functionele afhankelijkheden in relatie L, als LNAAM een kandidaat sleutel is (en WNPL → STATUS niet geldt).**

**6.4.2 Twee overlappende sleutels: een probleem.** Beschouw nu de relatie LLA, waarbij weer wordt aangenomen dat leveranciernamen uniek zijn.

LLA ( LNO, LNAAM, ANO, HOEV )

De kandidaatsleutels zijn (LNO,ANO) en (LNAAM,ANO). Is deze relatie in BCNV? Het antwoord is neen, omdat de relatie twee determinanten bevat, namelijk LNO en LNAAM, die geen kandidaatsleutels voor de relatie zijn. LNO is determinant, omdat hij functioneel LNAAM bepaalt; voor LNAAM geldt het omgekeerde. Een gedeeltelijke weergave van de relatie LLA zien we in Fig. 6.9.

LLA

| LNO | LNAAM | ANO | HOEV |
|---|---|---|---|
| L1 | Smit | A1 | 300 |
| L1 | Smit | A2 | 200 |
| L1 | Smit | A3 | 400 |
| L1 | Smit | A4 | 200 |

**Fig. 6.9 Extensie van relatie LLA.**

Zoals Fig. 6.9 weergeeft, heeft de relatie LLA dezelfde soort redundanties als de relaties EERSTE en TWEEDE van de vorige paragrafen. Dus LLA is onderworpen aan dezelfde soort bijwerkanomalieën. Het veranderen van de naam van leverancier L1 van Smit in Ruis leidt bijvoorbeeld òf tot zoekproblemen òf tot gevaar voor inconsistenties. Toch is LLA in 3NV, omdat in 3NV alleen de niet-sleutel

attributen volledig van de primaire sleutel moeten afhangen. Er wordt bij de 3NV niets gezegd over de situatie dat een attribuut zelf deel is van een alternatieve sleutel. Dus het feit dat LNAAM niet volledig van (LNO,ANO) afhangt, wordt genegeerd.

De oplossing tot het LLA probleem is natuurlijk het reduceren van de relatie. In dit geval gebeurt dit door de projecties:

LL ( LNO, LNAAM ) en LA ( LNO, ANO, HOEV )

of, het andere alternatief, de projecties

LL ( LNO, LNAAM ) en LA ( LNAAM, ANO, HOEV )

Al deze relaties zijn in BCNV.

Als tweede voorbeeld beschouwen we de relatie SOD met attributen S (student), O (onderwerp) en D (docent). De betekenis van een SOD tupel (s,o,d) is dat student s onderwerp o gedoceerd krijgt door docent d. Veronderstel bovendien dat de volgende restricties gelden.

- Voor ieder onderwerp wordt een student door slechts één docent in dat onderwerp onderwezen.
- Elke docent onderwijst slechts één onderwerp.
- Elk onderwerp wordt door vele docenten onderwezen.

Fig. 6.10 schetst een mogelijke inhoud van de relatie SOD.

| SOD | S | O | D |
|---|---|---|---|
| | Smit | Wiskunde | Prof. Witte |
| | Smit | Natuurkunde | Prof. Groen |
| | Janse | Wiskunde | Prof. Witte |
| | Janse | Natuurkunde | Prof. Bruin |

**Fig. 6.10 Extensie van relatie SOD.**

Wat zijn de FA'en in relatie SOD? De eerste restrictie levert ons de afhankelijkheid van D ten aanzien van de combinatie (S,O). De tweede restrictie bepaalt dat O functioneel afhankelijk is van D. Tenslotte houdt de derde restrictie in dat er geen afhankelijkheid van D ten aanzien van O is. Dus het FA diagram ziet er uit zoals in Fig. 6.11 is weergegeven. Opnieuw hebben we twee overlappende kandidaatsleutels, namelijk de combinatie (S,O) en de combinatie (S,D) (ga na of deze twee combinaties sleutels zijn). Natuurlijk is D alleen geen kandidaatsleutel, omdat hij S niet bepaalt. Opnieuw is de relatie niet in BCNV, immers D is determinant en *geen* kandidaatsleutel; opnieuw lijdt de relatie aan zekere bijwerkproblemen. Als we bijvoorbeeld de informatie dat Janse

natuurkunde studeert willen verwijderen, dan kunnen we dat niet bereiken zonder dat tegelijkertijd de informatie dat Prof. Bruin natuurkunde geeft wordt verwijderd.

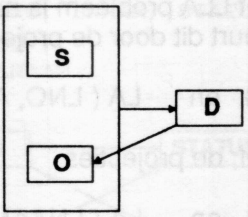



**Fig. 6.11 Functionele afhankelijkheden in de relatie SOD.**

De moeilijkheden ontstaan doordat D een determinant is, maar geen kandidaatsleutel. De remedie is weer dezelfde als altijd, namelijk het reduceren van de oorspronkelijke relatie tot projecties in BCNV, in dit geval de projecties:

SD ( S, D ) en DO ( D, O )

Het wordt aan de lezer overgelaten om de inhouden van deze twee relaties te geven, aangenomen dat SOD de inhoud van Fig. 6.10 heeft. Bovendien wordt aan de lezer overgelaten de FA diagrammen van de twee relaties SD en DO te geven (*Vraag:* Wat zijn in SD en DO de kandidaatsleutels?). In paragraaf 6.5 zullen we zien dat het reduceren van een relatie niet altijd zonder problemen gaat.

**6.4.3 Twee overlappende sleutels: geen probleem.** Tenslotte geven we een voorbeeld van een relatie met twee overlappende sleutels waarbij *geen* problemen optreden. Hiermee wordt geïllustreerd dat overlappende sleutels niet per definitie problemen inhouden. Beschouw de relatie EXAMEN met de attributen S (student), O (onderwerp) en P (plaats). De betekenis van een tupel (s,o,p) van de relatie EXAMEN is, dat student s voor onderwerp o een examen heeft afgelegd en daarbij de p-de plaats heeft behaald. Hierbij wordt de volgende restrictie geacht te gelden:

- Het kan niet voorkomen dat twee studenten dezelfde plaats voor hetzelfde examen halen.

Het FA diagram wordt in Fig. 6.12 getoond. Opnieuw zijn er twee overlappende sleutels, namelijk (S,O) en (O,P). De relatie is echter keurig in BCNV, omdat deze twee kandidaatsleutels de enige determinanten zijn. De lezer gaat na dat in de relatie EXAMEN geen

bijwerkanomalieën kunnen voorkomen. Hiermee is aangetoond dat overlappende sleutels niet altijd in problemen resulteren.

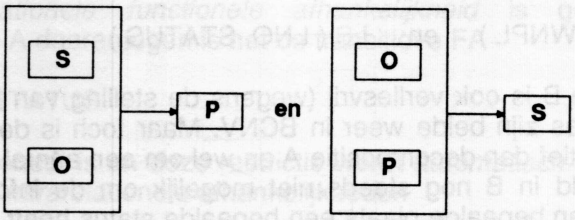



**Fig. 6.12 Functionele afhankelijkheden in de relatie EXAMEN.**

Concluderend zien we dat het begrip BCNV bepaalde problemen uit de weg ruimt waar het begrip 3NV geen raad mee weet. Bovendien is het begrip BCNV conceptueel eenvoudiger, omdat het de begrippen 1NV, 2NV, primaire sleutel en functionele afhankelijkheid niet gebruikt. Aan de andere kant zijn de begrippen primaire sleutel, transitieve afhankelijkheid, etc. om 'didactische' redenen belangrijk, omdat ze enig idee geven van het soort problemen dat zich kan voordoen als foutieve relaties worden gekozen.

## 6.5 GOEDE EN SLECHTE DECOMPOSITIES

Gedurende het reductieproces komt het vaak voor dat een gegeven relatie op meer dan één manier kan worden ontbonden. Beschouw de relatie TWEEDE van paragraaf 6.3 opnieuw. Daar golden de volgende FA'en:

TWEEDE.LNO -> TWEEDE.WNPL

en

TWEEDE.WNPL -> TWEEDE.STATUS

en daarom eveneens (door transitiviteit)

TWEEDE.LNO -> TWEEDE.STATUS

(zie Fig. 6.13, waar de transitieve afhankelijkheid door middel van een gestippelde pijl is weergegeven). We hebben in paragraaf 6.3 aangetoond dat de bijwerkanomalieën konden worden overwonnen als we TWEEDE vervingen door de twee 3NV projecties

LW ( LNO, WNPL ) en WS ( WNPL, STATUS )

(in feite zijn LW en WS niet alleen in 3NV, maar ook in BCNV). Laten we deze decompositie 'decompositie A' noemen. A is niet de enige decompositie, een tweede mogelijkheid zou decompositie B zijn:

LW ( LNO, WNPL ) en LS ( LNO, STATUS )

Decompositie B is ook verliesvrij (wegens de stelling van Heath) en de twee projecties zijn beide weer in BCNV. Maar toch is decompositie B minder attractief dan decompositie A en wel om een aantal redenen. Het is bijvoorbeeld in B nog steeds niet mogelijk om de informatie op te nemen dat een bepaalde plaats een bepaalde status heeft, tenzij er een leverancier in die plaats zijn domicilie heeft.

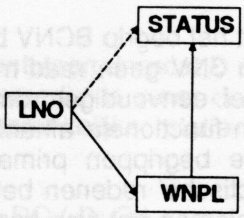



**Fig. 6.13 Functionele afhankelijkheden in de relatie TWEEDE.**

Laten we dit voorbeeld eens in meer detail bekijken. Op de eerste plaats merken we op dat de projecties van A overeenkomen met de *getrokken* pijlen van Fig. 6.13, terwijl een van de projecties van B overeenkomt met een *gestippelde* pijl (namelijk LS). In decompositie A zijn de twee projecties *onafhankelijk van elkaar* in de volgende betekenis: bijwerkingen kunnen in de ene relatie worden gedaan zonder rekening te houden met de andere. Aangenomen dat een dergelijke bijwerkactie legaal is in de context van de betrokken relatie (dat wil zeggen dat de entiteit-integriteitsregel voor die relatie niet overtreden wordt), dan zal *de join van die twee projecties na het bijwerken altijd weer een valide relatie TWEEDE opleveren* (dat wil zeggen, dat de join niet een van de FA'en van TWEEDE zal overtreden). In decompositie B zal het bijwerken van een van de twee relaties inhouden dat nog gekeken moet worden of de FA

TWEEDE.WNPL -> TWEEDE.STATUS

niet overtreden wordt. (Dit houdt in dat gekeken moet worden of leveranciers die in dezelfde plaats wonen ook dezelfde status hebben; beschouw bijvoorbeeld eens de acties die moeten geschieden als in decompositie B een leverancier van Londen naar Parijs verhuist.) Met andere woorden, in B zijn de twee projecties niet onafhankelijk.

Het echte probleem is dat in decompositie B de FA

WNPL -> STATUS

een *interrelationele functionele afhankelijkheid* is geworden. In decompositie A daarentegen is het de transitieve FA

LNO -> STATUS

die interrelationeel is en deze restrictie wordt *automatisch* afgedwongen door de twee *intra*relationele afhankelijkheden

LNO -> WNPL

en

WNPL -> STATUS.

Merk op dat in deze drie FA'en de relatie-prefixen zijn weggelaten.

Het begrip 'onafhankelijke projecties' helpt ons dus aan richtlijnen bij het kiezen van een decompositie als er meer dan één mogelijk is. In het algemeen is een decompositie waarin de projecties onafhankelijk zijn, te verkiezen boven een decompositie waarvoor dat niet geldt. Rissanen [RISS77] toont aan dat de projecties R1 en R2 van een relatie R onafhankelijk zijn in de betekenis als hiervoor omschreven, dan en slechts dan als

1. iedere FA in R logisch kan worden afgeleid van de FA'en in R1 en R2, en

2. de gemeenschappelijke attributen van R1 en R2 een kandidaatsleutel voor R1 of R2 vormen.

Rissanen's stelling maakt het controleren van onafhankelijkheid tot een eenvoudige taak. Beschouw twee decomposities A en B zoals hiervoor gedefinieerd. In A zijn de twee decomposities onafhankelijk, omdat hun gemeenschappelijk attribuut WNPL de primaire sleutel van WS is en omdat iedere FA van TWEEDE in een van de twee projecties verschijnt of een logisch gevolg is van de FA'en in de projecties. In B daarentegen zijn de twee projecties niet onafhankelijk, omdat de FA

WNPL -> STATUS

niet uit de FA'en van de projecties van B kan worden afgeleid (ofschoon het waar is dat hun gemeenschappelijk attribuut LNO de primaire sleutel van beide decomposities is).

We merken terzijde nog even op dat de derde decompositie, namelijk het vervangen van TWEEDE door de projecties

LS ( LNO, STATUS ) en WS ( WNPL, STATUS ),

niet valide is omdat de decompositie niet verliesvrij is. (Ga dit na!).

Een relatie die niet kan worden ontbonden in onafhankelijke componenten, wordt *atomair* genoemd. We suggereren niet, dat een niet-atomaire relatie noodzakelijkerwijs moet worden ontbonden in atomaire componenten; zo zijn bijvoorbeeld L en A van de leveranciers-artikelen database niet atomair, maar toch heeft het geen zin om ze verder te ontbinden. Relatie LA is daarentegen wel atomair.

Merk tenslotte op, dat een *atomaire* relatie soms *verliesvrij* kan worden ontbonden in *niet onafhankelijke componenten.* Beschouw de relatie SOD van paragraaf 6.4 met de FA'en

(S,O) -> D

en

D -> O

(we negeren opnieuw de relatie-prefixen). Zoals we gezien hebben, kan deze relatie (die in 3NV maar niet in BCNV is) verliesvrij worden ontbonden in de twee BCNV projecties

SD ( S, D ) en DO ( D, O ).

Echter, deze projecties zijn volgens Rissanen's stelling niet onafhankelijk, omdat de FA

(S,O) -> D

niet kan worden afgeleid van de FA

D -> O

en deze FA de enige is, die in de projecties voorkomt. Anders gezegd, het is de enige intra-relationele afhankelijkheid. Het gevolg is dat de twee projecties niet onafhankelijk van elkaar kunnen worden bijgewerkt (bewijs dit). Feitelijk is SOD atomair, ook al is hij niet in BCNV.

We worden gedwongen tot de onaangename conclusie dat de *twee doelstellingen* bij het *database-ontwerp*, namelijk het ontbinden van een relatie in BCNV componenten en het ontbinden in onafhankelijke componenten, *soms niet beide tegelijk haalbaar zijn.*

## 6.6 SAMENVATTING EN SLOTOPMERKINGEN

Uitgaande van een relatie in 1NV kan normalisatie als volgt worden samengevat.

1. De relatie heeft een samengestelde primaire sleutel, zeg van de vorm (S1,S2), en bevat ook een attribuut A dat functioneel afhankelijk is van een deel van de sleutel, bijvoorbeeld van attribuut S2 in plaats van van de hele sleutel (S1,S2). (De relatie EERSTE is een voorbeeld van zo'n geval.) In dit geval wordt aanbevolen om een relatie met als attributen S2 en A af te splitsen en om A uit de oorspronkelijke relatie te halen:

   **Vervang**
   R(S1,S2,A) met primaire sleutel (S1,S2)
   en met S2 --> A
   **door**
   R1(S1,S2) met primaire sleutel (S1,S2)
   **en**
   R2(S2,A) met primaire sleutel S2

2. De relatie heeft een primaire sleutel S, een attribuut A1 en een attribuut A2 dat functioneel afhankelijk is van A1 (A1 en A2 zijn natuurlijk functioneel afhankelijk van S). De oplossing is in wezen dezelfde als hierboven: we splitsen een relatie af die A1 en A2 bevat en waarvan A1 de primaire sleutel wordt. Verder verwijderen we A2 uit de oorspronkelijke relatie.

   **Vervang**
   R(S,A1,A2) met primaire sleutel S
   en met A1 --> A2
   **door**
   R1(S,A1) met primaire sleutel S
   **en**
   R2(A1,A2) met primaire sleutel A1.

3. De relatie heeft meerdere, samengestelde en overlappende kandidaatsleutels, zeg (G,S) en (G,T). Het attribuut G zit dus in beide kandidaatsleutels. *Bovendien* kent de relatie de functionele afhankelijkheid S -> T. De oplossing is dezelfde als hierboven: we splitsen een relatie met de attributen S,T af en schrappen T uit de oorspronkelijke relatie.

**Vervang** R(S,G,T) met kandidaatsleutels (G,S) en (G,T) en met S -> T
**door** R1(G,S) met primaire sleutel (G,S)
**en** R2(S,T) met primaire sleutel S.

Gegeven een relatie als invoer, dan zal een herhaald toepassen van genoemde richtlijnen ervoor zorgen dat een equivalente verzameling relaties in BCNV resulteert.

We merken op dat er naast de BCNV nog andere normaalvormen zijn, namelijk de vierde (4NV) en de vijfde normaalvorm (5NV), die 'vreemdere' situaties betreffen dan de situatie die door BCNV wordt vermeden. Deze 4NV en 5NV vallen buiten de scope van dit boek. De reden hiervoor is dat relaties die wel in BCNV zijn maar niet in 4NV of 5NV in de praktijk zeer weinig voorkomen, alhoewel dit theoretisch wel mogelijk is. Daarom spreekt men vaak over het verkrijgen van relaties in BCNV (of zelfs 3NV), alsof deze normaalvorm het uiteindelijk doel van normalisatie zou zijn. In de praktijk is deze vereenvoudiging, alhoewel theoretisch niet perfect, zeer wel te rechtvaardigen.

Het algemene doel van het reductieproces is om relaties te verkrijgen waarin alleen functionele afhankelijkheden tussen attributen en de sleutel(s) van een relatie bestaat(n). Bovendien wordt redundantie vermeden waardoor het principe van 'één feit op één plaats' overeind wordt gehouden. Zoals al vaker is beweerd, zijn de normalisatieregels slechts richtlijnen; soms zijn er goede redenen om er vanaf te wijken. Het klassieke voorbeeld is de naam-en-adres relatie:

NADRES ( NAAM, STRAAT, HUISNUMMER, POSTCODE, STAD)

waarin NAAM de primaire sleutel is en waarbij bovendien de volgende FA zou kunnen gelden:

(STRAAT, HUISNUMMER, STAD) -> POSTCODE

Deze relatie is niet in 3NV (er is immers een transitieve afhankelijkheid). De normalisatieregels zouden nu een decompositie

NSHS ( NAAM, STRAAT, HUISNUMMER, STAD)
SHSP ( STRAAT, HUISNUMMER, STAD, POSTCODE)

voorschrijven met respectievelijk NAAM en (STRAAT, HUISNUMMER, STAD) als primaire sleutels. De attributen STRAAT, HUISNUMMER, STAD en POSTCODE worden echter nagenoeg alleen tezamen gebruikt (denk aan adreslijsten). Bovendien wijzigen postcodes zelden of nooit (dus er is nauwelijks gevaar van inconsistentie). Daarom wordt deze decompositie vaak niet uitgevoerd.

## OPGAVEN

**6.1** Gegeven is de relatie R(A,B,C,D). Geef voor onderstaande gevallen a. tot en met e. het FA diagram en de normaalvorm waarin R zich bevindt.

a. AB -> C, AB -> D, C -> D.
b. AB -> D, A -> C.
c. AB -> C, AB -> D.
d. ABC -> D, D -> A.
e. ABC -> D, D -> A, D ->B, D -> C.

**6.2** Welke van de volgende beweringen is waar. Geef voor die beweringen die fout zijn een tegenvoorbeeld. Ga er vanuit dat iedere relatie tenminste in 1NV is.

a. Elke binaire relatie is in BCNV.
b. Elke relatie met een enkelvoudige sleutel is in 3NV.
c. Elke relatie met een enkelvoudige sleutel is in 2NV.
d. Relatie R (A,B,C) is gelijk aan de join van de projecties R1 (A,B) en R2 (A,C) als in R de FA A -> B geldt.
e. Als R.A -> R.B en R.B -> R.C, dan geldt ook dat R.C -> R.A.
f. Als R.A -> R.B en R.A -> R.C, dan geldt ook dat R.A -> R.(B, C).
g. Als R.B -> R.A en R.C -> R.A, dan geldt ook dat R.(B,C) -> R.A.
h. Als R.B -> R.A dan geldt ook dat R.(B,C) -> R.A.

**6.3** Voor een relatie R(A,B,C,D) gelden de volgende FA'en:
A -> B, A -> C, A -> D
B -> A, B -> C, B -> D
In welke normaalvorm is R?

**6.4** Gegeven de relatie R(A,B,C,D,E) en de volgende functionele afhankelijkheden:
A -> B, A -> D, A -> E,
(B,C) -> E, C -> E, C -> A.
Teken het FA diagram. Welke FA'en kunnen van andere afgeleid worden? In welke normaalvorm is deze relatie? Indien R niet in BCNV is, transformeer R dan in een aantal relaties die wel in BCNV zijn.

**6.5** De relatie R heeft de attributen A, B, C, D en E. De volgende functionele afhankelijkheden zijn gegeven:
(A,B) -> (C,D,E), B -> D, D -> C.
a. Welke FA'en kunnen van de andere afgeleid worden? Geef het resterende FA diagram.
b. Ontbind de relatie R op grond van het FA diagram in een aantal relaties in BCNV.
c. Neem aan dat bovendien geldt D -> B. Ontbind de relatie met deze extra FA.
Motiveer de stappen in uw decompositie algoritme.

**6.6** Op een conferentie worden door sprekers vele voordrachten gehouden. Een spreker houdt gedurende de conferentie slechts één voordracht. Een relatie PLANNING heeft de volgende attributen.

| | |
|---|---|
| T | Tijdstip |
| Z | Nummer van het zaal |
| S | Naam van de spreker |
| D | Naam van de deelnemer |
| V | Titel van de voordracht |

Een tupel <t,z,s,d,v> is een element van deze relatie als op het tijdstip t in zaal z de voordracht v start met s als spreker en d als deelnemer. Gevraagd wordt om de FA'en op te stellen en het bijbehorende FA diagram te geven. Gebruik deze informatie om PLANNING tot een meer gewenste structuur te reduceren.

**6.7** Een opleidingsinstituut gebruikt voor de planning van cursussen een gegevensbank. De cursussen hebben een unieke naam en een uniek cursusnummer. Het instituut heeft alle docenten een docentnummer gegeven. Van een docent is verder bekend: naam, adresgegevens en welke cursussen hij bereid is te geven. Om de opleidingen zo goed mogelijk te verzorgen, houdt het opleidingsinstituut de geschiktheid van een docent voor het geven van een bepaalde cursus bij. Er wordt een planning gemaakt van de cursussen. Een geplande cursus wordt voorzien van een uniek plannummer. De plaats waar de cursus wordt gegeven en de aanvangsdatum worden vastgelegd. Een geplande cursus gaat niet door indien het aantal aanmeldingen kleiner is dan het voor die cursus geldende minimum. Het maximum aantal deelnemers aan een geplande cursus hangt af van de plaats waar de cursus wordt gegeven. Voor elke cursus wordt een docent gezocht.
Geef de entiteittypen, associatietypen en niet atomaire kenmerken die in de probleemstelling voorkomen. Ontwerp een relationele database, waarin alle onderstreepte kenmerken (attributen) voorkomen en waarvan de relaties in BCNV zijn. Geef de primaire en vreemde sleutels aan via onderstreping respectievelijk cursivering (of stippellijn).

**6.8** In een database worden gegevens betreffende studenten bijgehouden. Van een student zijn naam (uniek), adres en woonplaats geregistreerd. Studenten worden 'modulair' onderwezen, dat wil zeggen dat het onderwijs in vele modules is onderverdeeld. Iedere student kiest het pakket modules dat hij wil gaan volgen. Een pakket heeft een zekere status (bijvoorbeeld goedgekeurd). Per module worden de code (uniek), de instituten waar die module wordt onderwezen en de zwaarte van het module opgenomen. Een instituut wordt door een unieke naam geïdentificeerd en kent naast een secretarieel adres ook nog een aantal adressen waar het modulaire onderwijs plaats vindt. Voor elk instituut geldt dat een module slechts op één (onderwijs)adres wordt gegeven. Een student mag voor een bepaald module meer dan een keer tentamen doen; op een tentamen volgt een beoordeling.

Beantwoord dezelfde vragen als in 6.7.

# 7
# Manipulatie van een Relationele Database

## 7.1 INLEIDING

In hoofdstuk 3 zijn we kort ingegaan op de operatoren in een relationele algebra. De belangrijkste eigenschap van deze operatoren is dat ze in plaats van op één tupel op een verzameling van tupels opereren. Deze relationele algebra werd al in paragraaf 3.5 ingevoerd, omdat de begrippen join, projectie en dergelijke een grote rol spelen bij normalisatie. In de hoofdstukken 4, 5 en 6 hebben we ons geconcentreerd op de gegevenskant van het relationele data model. In dit hoofdstuk zullen we onze aandacht richten op SQL, een taal die ontwikkeld is door Chamberlin en anderen aan het IBM Research Laboratory in San José, Californië [CHAM76]. (Een volledige beschrijving van SQL vindt men in [VDLA87] en [DATE84].) Een prototype van de taal werd eerder ontwikkeld in het kader van het "SYSTEM R" project aldaar.

SQL is een taal van het externe of conceptuele niveau. SQL voorziet in vier DML opdrachten, namelijk SELECT, UPDATE, DELETE en INSERT. In dit hoofdstuk worden de belangrijkste karakteristieken van deze vier opdrachten besproken. De SELECT wordt gebruikt om gegevens aan de database te onttrekken, UPDATE neemt wat voorheen WIJZIGEN werd genoemd voor zijn rekening, DELETE verzorgt het VERWIJDEREN en INSERT het INVOEGEN.

SQL werkt niet op relaties, doch op daarmee verwante tabellen. Een belangrijk verschil tussen relaties en tabellen is, dat relaties geen duplicaten kennen en tabellen wel. De opdracht in SQL waarmee tabellen worden gecreëerd, heet CREATE TABLE. Deze opdracht is feitelijk het SQL analogon van de RELATION opdracht zoals die in hoofdstuk 4 is ingevoerd. Een ander verschil is dat in SQL attributen

kolommen worden genoemd en tupels rijen, om de 'tabel-layout' van SQL te benadrukken en om de termen tupel en attribuut gereserveerd te houden voor relaties. De indeling van dit hoofdstuk is als volgt:

- In paragraaf 7.2 worden de belangrijkste kenmerken van de SELECT opdracht geïntroduceerd.
- Paragraaf 7.3 laat zien hoe de SELECT opdracht kan worden gebruikt voor de vele soorten joins (in het bijzonder natuurlijk de zogenaamde *natuurlijke join*).
- Daarna komen in paragraaf 7.4 enige *ingebouwde functies* zoals COUNT, SUM en AVG aan de ordes; in het bijzonder wordt het gebruik van de GROUP BY en HAVING-clausules in samenhang met deze ingebouwde functies besproken.
- In 7.5 komen nog enige extra faciliteiten aan de orde, namelijk de NULL-conditie, subvragen en de EXISTS-clausule. Deze taalelementen, ofschoon ze in sommige gevallen zeer belangrijk zijn, worden in de praktijk minder gebruikt dan de in 7.2 tot 7.4 ingevoerde taalelementen en worden daarom in een aparte paragraaf opgenomen.
- Bijwerkoperaties worden in 7.6 behandeld.
- Tenslotte wordt in paragraaf 7.7 geschetst hoe een complexe vraag uitgevoerd zou kunnen worden. In deze paragraaf wordt een indruk gegeven van de zogenaamde conceptuele afhandeling van een vraag, dat wil zeggen, de echte afhandeling hoeft niet zo te geschieden (bijvoorbeeld omdat de conceptuele afhandeling veel te traag zou zijn).

L

| LNO | LNAAM | STATUS | WNPL |
|---|---|---|---|
| L1 | Smit | 20 | Londen |
| L2 | Janse | 10 | Parijs |
| L3 | Blaak | 30 | Parijs |
| L4 | Klerk | 20 | Londen |
| L5 | Adams | 30 | Athene |

A

| A | ANAAM | KLEUR | GEW. | WNPL |
|---|---|---|---|---|
| A1 | Moer | Rood | 12 | Londen |
| A2 | Bout | Groen | 17 | Parijs |
| A3 | Schroef | Blauw | 17 | Rome |
| A4 | Schroef | Rood | 14 | Londen |
| A5 | Kamrad | Blauw | 12 | Parijs |
| A6 | Nok | Rood | 19 | Londen |

LA

| L | A | QTY |
|---|---|---|
| L1 | A1 | 300 |
| L1 | A2 | 200 |
| L1 | A3 | 400 |
| L1 | A4 | 200 |
| L1 | A5 | 100 |
| L1 | A6 | 100 |
| L2 | A1 | 300 |
| L2 | A2 | 400 |
| L3 | A2 | 200 |
| L4 | A2 | 200 |
| L4 | A4 | 300 |
| L4 | A5 | 400 |

**Fig. 7.1 De leveranciers - artikelen database.**

Zoals gebruikelijk wordt de leveranciers-artikelen database als voorbeeld database genomen (zie Fig. 7.1 voor de relationele visie op de database). Er wordt in dit hoofdstuk niet gepoogd om een alles omvattende behandeling van SQL te geven; alleen de belangrijkste kenmerken van SQL worden behandeld. Het zal blijken dat veel SQL formuleringen tamelijk complex zijn. Hieruit moet niet geconcludeerd worden dat SQL op zich complex is. Het punt is dat in SQL (en in de meeste relationele talen) 'gewone' problemen zeer eenvoudig op te lossen zijn, zodat eenvoudige voorbeelden niet interessant zijn en niet de volle kracht van de taal tonen. Natuurlijk zullen we ook wel eenvoudige voorbeelden geven.

## 7.2 EENVOUDIGE VRAGEN

We beginnen met een eenvoudig probleem, namelijk met de formulering van de vraag "Geef de nummers en de status van de Parijse leveranciers". Dit wordt in SQL:

```
SELECT   LNO, STATUS
FROM     L
WHERE    WNPL = 'Parijs'
```

*Resultaat:*

```
LNO STATUS
----------
L2    10
L3    30
```

Dit voorbeeld illustreert de eenvoudigste vorm van de SQL SELECT opdracht. SELECT specificeert kolommen (in relationele termen, de attributen), FROM specificeert een tabel (in relationele termen een relatie) en WHERE de conditie waaraan de rijen moeten voldoen willen ze tot het resultaat bijdragen. Het eerste punt dat benadrukt moet worden, is dat *het resultaat van een vraag weer een tabel is*; namelijk een tabel die op de een of andere wijze van de bestaande tabellen in de database is afgeleid. Met andere woorden, de gebruiker van een relationeel systeem werkt altijd in een eenvoudig tabellarisch raamwerk, hetgeen een erg aangename eigenschap van deze systemen is. Vanwege dit feit zeggen we dat relationele tabellen een *gesloten systeem* over de opzoek-operatoren vormen. Deze eigenschap van relationele systemen noemen we de *geslotenheidseigenschap*. (De Engelse term voor opzoeken is retrieval.)

We hadden de bovenstaande vraag ook kunnen formuleren met gebruikmaking van gekwalificeerde kolomnamen:

```
SELECT   L.LNO, L.STATUS
FROM     L
WHERE    L.WNPL = 'Parijs'
```

Een gekwalificeerde kolomnaam bestaat uit een tabelnaam en een kolomnaam (in die volgorde), die worden gescheiden door een punt. Het is soms noodzakelijk, met name om dubbelzinnigheid te voorkomen, om gekwalificeerde kolomnamen te gebruiken; fout is het echter nooit!

De basisvorm van de SELECT opdracht wordt hieronder gegeven. Hierbij zijn vele details, waarvan de meeste in dit hoofdstuk nog wel zullen worden uitgelegd, genegeerd; een iets completere syntax van de behandelde SQL faciliteiten vindt men in Appendix A, waar ook de betekenis van de gebruikte metasymbolen wordt uitgelegd. Voor een volledige beschrijving van de SQL syntax verwijzen we naar [VDLA87].

*SQL-opdracht*

```
  SELECT     [ DISTINCT ] { { kolom-expressie } ... | * }
  FROM       { tabelnaam  [ alias ] }  ...
[ WHERE      conditie ]
[ GROUP BY   { kolomspecificatie }  ...
[ HAVING     conditie ] ]
[ ORDER BY   { kolomspecificatie  sorteervolgorde } ... ]
```

*kolom-expressie*

```
{ tabelnaam.* | expressie | functie }
```

*expressie*

```
{ kolomspecificatie | constante }
```

We gaan nu door middel van een aantal voorbeelden de belangrijkste karakteristieken van de SELECT opdracht uitleggen.

**7.2.1 Opzoeken.** Geef de artikelnummers van de leverbare artikelen.

```
SELECT   ANO
FROM     LA
```

*Resultaat:*

```
ANO
---
A1
A2
A3
A4
A5
A6
A1
A2
A2
A2
A4
A5
```

Merk op dat er in het resultaat duplicaten voorkomen. SQL elimineert de duplicaten van het resultaat van een SELECT opdracht niet, tenzij de gebruiker hier expliciet via het sleutelwoord DISTINCT om vraagt. Bijvoorbeeld:

```
SELECT   DISTINCT ANO
FROM     LA
```

*Resultaat:*

```
ANO
---
A1
A2
A3
A4
A5
A6
```

**7.2.2 Sorteren en uitdrukkingen.** Geef voor alle artikelen het artikelnummer en het gewicht van het artikel in grammen (in tabel A zijn de gewichten in Engelse ponden gegeven).

```
SELECT   A.ANO, 'gewicht in grammen =', A.GEW * 454
FROM     A
ORDER BY A.ANO ASC
```

*Resultaat:*

```
ANO
-----------------------------------
A1    Gewicht in grammen =    5448
A2    Gewicht in grammen =    7718
A3    Gewicht in grammen =    7718
A4    Gewicht in grammen =    6356
A5    Gewicht in grammen =    5448
A6    Gewicht in grammen =    8626
```

In het algemeen kunnen in de SELECT-clausule willekeurige uitdrukkingen staan. Deze uitdrukkingen kunnen kolomnamen en/of constanten bevatten, die door de gebruikelijke aritmetische operatoren (+,-,*,/) worden samengevoegd en waarbij optionele haakjes de gewenste evaluatievolgorde aangeven. Zulke uitdrukkingen kunnen ook ingebouwde functies bevatten (zie paragraaf 7.4).

De ORDER BY clausule bewerkstelligt dat de uitvoer wordt gesorteerd. Naast "ASC" (ascending, opklimmend) is ook "DESC" (descending, dalend) mogelijk.

**7.2.3 Opzoeken van hele tabellen.** Geef de volledige details van alle leveranciers.

```
SELECT    *
FROM      L
```

Resultaat: Een kopie van de hele L tabel. De '*' is een verkorte notatie voor de rij van alle kolomnamen van de tabel(len) in de FROM-clausule. De volgorde wordt bepaald door de overeenkomstige CREATE TABLE opdracht (het SQL analogon van de RELATION opdracht van hoofdstuk 4). De SELECT opdracht is dus equivalent met:

```
SELECT    L.LNO, L.LNAAM, L.STATUS, L.WNPL
FROM      L
```

De '*' notatie is gemakkelijk voor de eindgebruiker, omdat toetsaanslagen worden uitgespaard.

**7.2.4 Opzoeken met condities.** Geef de nummers van de Parijse leveranciers die een status > 20 hebben.

```
SELECT    LNO
FROM      L
WHERE     WNPL = 'Parijs'
AND       STATUS > 20
```

*Resultaat:*

```
LNO
---
L3
```

De *conditie* in de WHERE-clausule mag bevatten:
- vergelijkingsoperatoren =,~=,>,~>,>=,<,~<,<=, (waarbij "~=" "niet gelijk" betekent, etc.),
- booleaanse operatoren AND, OR en NOT, en
- haakjes om de gewenste evaluatievolgorde aan te geven.

## 7.3 JOIN VRAGEN

Het vermogen om twee of meer tabellen met elkaar te verbinden is een van de krachtigste eigenschappen van relationele systemen. De operatie die dit bewerkstelligt, wordt 'join' genoemd (zie paragraaf 3.5). In feite is het de beschikbaarheid van de join-operator, meer dan iets anders, die de relationele van de niet-relationele systemen onderscheidt. Wat is nu een join? Eenvoudig gezegd is het een *vraag waarmee gegevens uit meer dan één tabel worden gehaald.* Hier volgt een eenvoudig voorbeeld.

**7.3.1 Eenvoudige equijoin.** Geef alle combinaties van leverancier- en artikelinformatie zodanig dat de leverancier en het artikel in dezelfde plaats zijn gelokaliseerd.

```
SELECT   L.*, A.*
FROM     L, A
WHERE    L.WNPL = A.PLAATS
```

*Resultaat:*

| LNO | LNAAM | STATUS | WNPL | ANO | ANAAM | KLEUR | GEW | PLAATS |
|---|---|---|---|---|---|---|---|---|
| L1 | Smit | 20 | Londen | A1 | Moer | Rood | 12 | Londen |
| L1 | Smit | 20 | Londen | A4 | Schroef | Rood | 14 | Londen |
| L1 | Smit | 20 | Londen | A6 | Kamrad | Rood | 19 | Londen |
| L2 | Janse | 10 | Parijs | A2 | Bout | Groen | 17 | Parijs |
| L2 | Janse | 10 | Parijs | A5 | Nok | Blauw | 12 | Parijs |
| L3 | Blaak | 30 | Parijs | A2 | Bout | Groen | 17 | Parijs |
| L3 | Blaak | 30 | Parijs | A5 | Nok | Blauw | 12 | Parijs |
| L4 | Klerk | 20 | Londen | A1 | Moer | Rood | 12 | Londen |
| L4 | Klerk | 20 | Londen | A4 | Schroef | Rood | 14 | Londen |
| L4 | Klerk | 20 | Londen | A6 | Kamrad | Rood | 19 | Londen |

Uitleg: Uit de formulering in het Nederlands blijkt al dat de uitvoergegevens van twee tabellen moeten komen, namelijk van L en A. In de SQL formulering van de vraag noemen we daarom eerst deze twee tabellen in de FROM-clausule en daarna drukken we de verbinding tussen deze twee tabellen in de WHERE-clausule uit. Dit houdt in dat moet gelden L.WNPL = A.PLAATS. Om te begrijpen hoe dit werkt, beschouwen we het volgende tweetal rijen (een van iedere tabel):

```
LNO  LNAAM STATUS  WNPL        ANO ANAAM KLEUR GEW  PLAATS
-----------------------        ---------------------------
L1   Smit   20    Londen        A1  Moer  Rood  12  Londen
```

Deze twee rijen tonen aan dat leverancier L1 en artikel A1 inderdaad in dezelfde plaats gelokaliseerd zijn (immers WNPL en PLAATS zijn beide gelijk aan Londen). Uit deze twee rijen, die aan de equijoin-conditie voldoen, wordt dan de volgende rij gegenereerd:

```
LNO LNAAM STATUS  WNPL  ANO  ANAAM  KLEUR GEW  PLAATS
-------------------------------------------------------
L1  Smit   20    Londen  A1   Moer   Rood  12   Londen
```

Hetzelfde proces wordt uitgevoerd voor de andere rij-paren die voor WNPL en PLAATS gelijke waarden hebben. Merk op dat leverancier L5 (die in Athene is gevestigd) niet in het resultaat voorkomt, omdat er geen artikelen in Athene zijn opgeslagen. Evenzo verschijnt artikel A3 (opgeslagen in Rome) niet in het resultaat, omdat er geen leveranciers in Rome gevestigd zijn.

Het resultaat van deze vraag wordt een *join* van de tabellen L en A over gelijke WNPL en PLAATS waarden genoemd. De term 'join' wordt ook gebruikt om de operatie aan te duiden die de constructie van het resultaat bewerkstelligt. De conditie L.WNPL = A.PLAATS wordt een *join-conditie* genoemd.

De vergelijkingsoperator in de join-conditie behoeft geen gelijkheid te zijn, ofschoon dit vaak het geval zal zijn. Voorbeeld 7.3.2 laat zien dat een 'groter dan' join ook voorkomt. Als de operator echter gelijkheid is, dan wordt de join soms expliciet aangeduid met de term '*equijoin*'. De equijoin moet per definitie een resultaat produceren waarbij twee kolommen identieke waarden hebben (zie ook het voorbeeld hiervoor). Als een van de twee kolommen wordt geëlimineerd, dan wordt het resultaat een *natuurlijke join* genoemd. Hier volgt een SQL voorbeeld van een natuurlijke join (de natuurlijke join van L en A):

```
SELECT   LNO, LNAAM, STATUS, WNPL,
         ANO, ANAAM, KLEUR, GEW
FROM     L, A
WHERE    L.WNPL = A.PLAATS
```

De natuurlijke join is waarschijnlijk de nuttigste join, zodat de ongekwalificeerde term 'join' vaak wordt gebezigd om de natuurlijke join aan te duiden.

Het nu volgende beschrijft een alternatieve (en handige) manier om joins conceptueel te construeren. Allereerst wordt het Cartesische produkt gevormd van de in de FROM-clausule genoemde tabellen. Het Cartesisch produkt van een verzameling van n tabellen is de tabel bestaande uit alle mogelijke rijen t, zodat t de aaneenschakeling is van een rij van de eerste tabel, een rij van de tweede tabel,..., en een rij van de n-de tabel. Bijvoorbeeld, het Cartesisch produkt van de tabellen L en A (in die volgorde) is de volgende tabel:

| LNO | LNAAM | STATUS | WNPL | ANO | ANAAM | KLEUR | GEW | PLAATS |
|---|---|---|---|---|---|---|---|---|
| L1 | Smit | 20 | Londen | A1 | Moer | Rood | 12 | Londen |
| L1 | Smit | 20 | Londen | A2 | Bout | Groen | 17 | Londen |
| L1 | Smit | 20 | Londen | A3 | Schroef | Blauw | 17 | Londen |
| L1 | Smit | 20 | Londen | A4 | Schroef | Rood | 14 | Londen |
| L1 | Smit | 20 | Londen | A5 | Nok | Rood | 12 | Londen |
| L1 | Smit | 20 | Londen | A6 | Kamrad | Blauw | 19 | Londen |
| L2 | Janse | 10 | Parijs | A1 | Moer | Rood | 12 | Londen |
| . | . | . | . | . | . | . | . | . |
| . | . | . | . | . | . | . | . | . |
| . | . | . | . | . | . | . | . | . |
| L5 | Adams | 30 | Athene | A6 | Kamrad | Rood | 19 | Londen |

(De volledige tabel bevat 5 * 6 rijen = 30 rijen.) Nu elimineren we van dit Cartesisch produkt al die rijen die niet aan de join-conditie voldoen. Wat dan overblijft is het gewenste join-resultaat. In het genoemde voorbeeld verwijderen we alle rijen waarvoor niet geldt dat L.WNPL = A.PLAATS; wat resteert is precies de equijoin die we al eerder toonden.

**7.3.2 Groter-dan join.** Geef alle combinaties van leverancier- en artikelinformatie, zodanig dat de naam van de vestigingsplaats van de leverancier alfabetisch na de naam van de opslagplaats van het artikel komt.

```
SELECT   L.*, A.*
FROM     L, A
WHERE    L.WNPL > A.PLAATS
```

*Resultaat:*

| LNO | LNAAM | STATUS | WNPL | ANO | ANAAM | KLEUR | GEW | PLAATS |
|---|---|---|---|---|---|---|---|---|
| L2 | Janse | 10 | Parijs | A1 | Moer | Rood | 12 | Londen |
| L2 | Janse | 10 | Parijs | A4 | Schroef | Rood | 14 | Londen |
| L2 | Janse | 10 | Parijs | A6 | Kamrad | Rood | 19 | Londen |
| L3 | Blaak | 30 | Parijs | A1 | Moer | Rood | 12 | Londen |
| L3 | Blaak | 30 | Parijs | A4 | Schroef | Rood | 14 | Londen |
| L3 | Blaak | 30 | Parijs | A6 | Kamrad | Rood | 19 | Londen |

**7.3.3 Join met een extra conditie.** Geef alle combinaties van leverancier- en artikelinformatie waarvoor geldt dat de vestigingsplaats (van de leverancier) en de opslagplaats (van het artikel) gelijk zijn, maar sluit hierbij de leveranciers met status 20 uit.

```
SELECT   L.*, A.*
FROM     L, A
WHERE    L.WNPL = A.PLAATS
AND      L.STATUS ~= 20
```

*Resultaat:*

| LNO | LNAAM | STATUS | WNPL | ANO | ANAAM | KLEUR | GEW | PLAATS |
|---|---|---|---|---|---|---|---|---|
| L2 | Janse | 10 | Parijs | A2 | Bout | Groen | 17 | Parijs |
| L2 | Janse | 10 | Parijs | A5 | Nok | Blauw | 12 | Parijs |
| L3 | Blaak | 30 | Parijs | A2 | Bout | Groen | 17 | Parijs |
| L3 | Blaak | 30 | Parijs | A5 | Nok | Blauw | 12 | Parijs |

De WHERE-clausule in een join kan andere condities bevatten dan alleen maar de join-conditie zelf.

**7.3.4 Opzoeken van gespecificeerde kolommen van een join.** Geef alle combinaties van leveranciernummer/artikelnummer zodanig dat vestigingsplaats en opslagplaats gelijk zijn.

```
SELECT    L.LNO, A.ANO
FROM      L, A
WHERE     L.WNPL = A.PLAATS
```

*Resultaat:*

```
LNO   ANO
--------
L1    A1
L1    A4
L1    A6
L2    A2
L2    A5
L3    A2
L3    A5
L4    A1
L4    A4
L4    A6
```

Het is natuurlijk mogelijk om bepaalde kolommen uit een join-resultaat te selecteren. Dat wordt in bovenstaand voorbeeld geïllustreerd.

**7.3.5 Join van drie tabellen.** Geef alle <WNPL,PLAATS> paren, zodanig dat door een leverancier, gevestigd in WNPL, een artikel, opgeslagen in PLAATS, kan worden geleverd. Bijvoorbeeld, door leverancier L1 kan artikel A1 worden geleverd; leverancier L1 is gevestigd te Londen en artikel A1 is in Londen opgeslagen. Dan is <Londen,Londen> een paar van het resultaat.

```
SELECT    DISTINCT L.WNPL, A.PLAATS
FROM      L, LA, A
WHERE     L.LNO = LA.LNO
AND       LA.ANO = A.ANO
```

*Resultaat:*

```
WNPL      PLAATS
---------------
Londen    Londen
Londen    Parijs
Londen    Rome
Parijs    Londen
Parijs    Parijs
```

Er is geen principiële beperking op het aantal tabellen dat via een join samengevoegd kan worden. Merk op, dat DISTINCT gebruikt wordt om de duplicaten te verwijderen.

**7.3.6 Join van een tabel met zichzelf.** Geef alle paren van leveranciernummers, zodanig dat de twee bijbehorende leveranciers in dezelfde plaats gevestigd zijn.

```
SELECT   FIRST.LNO, SECOND.LNO
FROM     L FIRST, L SECOND
WHERE    FIRST.WNPL = SECOND.WNPL
AND      FIRST.LNO < SECOND.LNO
```

*Resultaat:*

```
LNO   LNO
---   ---
L1    L4
L2    L3
```

Deze vraag betreft een join van de L tabel met zichzelf. De L tabel verschijnt daarom tweemaal in de FROM-clausule. Om deze twee verschijningsvormen te onderscheiden, introduceren we in die clausule twee willekeurige aliassen "FIRST" en "SECOND". Deze aliassen worden gebruikt om in de SELECT en de WHERE-clausule de kolomnamen uniek te maken. Merk op, dat het nut van de conditie FIRST.LNO < SECOND.LNO tweeledig is, namelijk (a) de paren leveranciernummers van de vorm (x,x) worden zo geëlimineerd en (b) er wordt vermeden dat zowel het paar (x,y) alsook (y,x) in de uitvoer verschijnen.

Dit is het eerste voorbeeld waarin het gebruik van aliassen noodzakelijk is. Vaak is het gebruik van aliassen niet noodzakelijk, maar wordt de formulering van de opdracht er wel duidelijker door.

## 7.4 INGEBOUWDE FUNCTIES

Ofschoon de tot nu toe beschreven SELECT opdracht tamelijk krachtig is, is hij toch nog niet adequaat voor vele praktische problemen. Zo kan bijvoorbeeld een eenvoudige vraag als "Hoeveel leveranciers zijn er?" niet worden uitgedrukt in de tot nu toe ingevoerde taalelementen. SQL biedt daarom een aantal *ingebouwde functies* aan. Deze functies zijn COUNT, SUM, AVG, MAX en MIN. Met uitzondering van het speciale geval "COUNT(*)" (zie hieronder) werkt ieder van deze functies op de verzameling waarden in een kolom van een bepaalde tabel. Deze tabel mag een zogenaamde afgeleide tabel zijn, dat wil zeggen, een tabel die geconstrueerd is als het resultaat van een vraag (zie paragraaf 7.5.9).

Het resultaat van de toepassing van een functie is een enkelvoudige waarde, die op de volgende wijze is gedefinieerd:

- COUNT - aantal waarden van een kolom
- SUM - som van de waarden van een kolom
- AVG - gemiddelde van de waarden van een kolom
- MAX - grootste waarde van een kolom
- MIN - kleinste waarde van een kolom

Voor SUM en AVG moet de kolom numerieke waarden bevatten. In het algemeen mag het argument van de functie worden voorafgegaan door het sleutelwoord DISTINCT, om aan te geven dat (redundante) duplicaatwaarden moeten worden verwijderd voordat de functie wordt toegepast. Voor de functies MAX en MIN heeft DISTINCT natuurlijk geen effect. Voor COUNT *moet* DISTINCT worden gespecificeerd; de speciale functie COUNT(*) (waarbij DISTINCT *niet is toegestaan*) wordt aangeboden om alle rijen in een tabel te kunnen tellen zonder dat duplicaten worden verwijderd.

Alle null waarden in het argument worden eerst verwijderd voordat de functie wordt toegepast, ongeacht of DISTINCT is gespecificeerd of niet. Dit geldt niet in het geval van COUNT(*), waar null waarden worden behandeld als niet-null waarden. Als het argument (de verzameling waarden die op een zeker moment door de kolom worden aangenomen) de lege verzameling blijkt te zijn, dan is het resultaat van een COUNT de waarde nul (met één *l*); alle andere functies retourneren dan de waarde null (met twee *l's*).

**7.4.1 Functie in de SELECT-clausule.** Geef het aantal leveranciers.

```
SELECT   COUNT(*)
FROM     L
```

*Resultaat:*

```
---
 5
```

**7.4.2 Functie met DISTINCT in de SELECT-clausule.** Geef het aantal leveranciers dat artikelen kan leveren.

```
SELECT   COUNT (DISTINCT LNO)
FROM     LA
```

*Resultaat:*

```
---
 4
```

**7.4.3 Functie in de SELECT-clausule met een niet-lege WHERE clausule.** Geef het aantal leveranciers dat artikel A2 kan leveren.

```
SELECT    COUNT (*)
FROM      LA
WHERE     ANO = 'A2'
```

*Resultaat:*

```
---
 4
```

**7.4.4 Functie in de SELECT-clausule met een niet-lege WHERE clausule.** Bepaal de totale hoeveelheid van het leverbare artikel A2.

```
SELECT    SUM (HOEV)
FROM      LA
WHERE     ANO = 'A2'
```

*Resultaat:*

```
----
1000
```

**7.4.5 Gebruik van GROUP BY.** Voorbeeld 7.4.4 liet zien hoe de totale hoeveelheid van een leverbaar artikel te berekenen is. Veronderstel nu eens, dat het gewenst is om de totale hoeveelheid van ieder leverbaar artikel te berekenen, dat wil zeggen, voor ieder leverbaar artikel moet het artikelnummer met de totale hoeveelheid gegeven worden.

```
SELECT    ANO, SUM (HOEV)
FROM      LA
GROUP     BY ANO ;
```

*Resultaat:*

```
ANO
--------
A1    600
A2   1000
A3    400
A4    500
A5    500
A6    100
```

Uitleg: De GROUP BY operator voert conceptueel een herschikking van de tabel genoemd in de FROM-clausule uit, zodat zogenaamde partities of *groepen* ontstaan. Zo'n groep heeft de eigenschap dat alle rijen dezelfde waarde hebben voor de kolom die in de GROUP BY genoemd staat. Deze herschikking is conceptueel, dat wil zeggen dat de tabel niet fysiek gesorteerd of herschikt hoeft te worden. In het voorbeeld wordt de tabel gegroepeerd, zodanig dat er een groep met leveranciers van artikel A1 ontstaat, een voor artikel A2, etc. Dan wordt de SELECT-clausule toegepast op iedere groep afzonderlijk (in plaats van op iedere rij van de oorspronkelijk tabel). Elke uitdrukking in de SELECT-clausule *moet een enkelvoudige waarde per groep afleveren.* Dit kan op een van de volgende wijzen:

- de waarde van de GROUP BY kolom zelf (die voor de groep natuurlijk constant is) of een aritmetische uitdrukking van die waarde,
- een constante, of
- een functie zoals SUM die opereert op de waarden van een gegeven kolom in de groep en die een enkelvoudige waarde aflevert.

**7.4.6 Gebruik van HAVING.** Geef de nummers van de artikelen die door meer dan één leverancier geleverd kunnen worden.

```
SELECT    ANO
FROM      LA
GROUP     BY ANO
HAVING    COUNT (*) > 1
```

*Resultaat:*

```
ANO
---
A1
A2
A4
A5
```

HAVING is voor groepen wat de WHERE-clausule is voor rijen. (Als er een HAVING-clausule is gespecificeerd, dan moet GROUP BY ook gespecificeerd zijn.) Met andere woorden, HAVING wordt gebruikt om groepen te elimineren zoals de WHERE-clausule wordt gebruikt om rijen te elimineren. De uitdrukkingen in een HAVING-clausule moeten per groep slechts *één* waarde afleveren.

## 7.5 GEVORDERDE ASPECTEN

In deze paragraaf laten we diverse aspecten van de SELECT opdracht zien, die we bij gebrek aan beter maar "gevorderd" hebben genoemd. Deze aspecten worden in het algemeen minder gebruikt dan die van de paragrafen 7.2-7.4, alhoewel ze niet gecompliceerd en bovendien soms zelfs fundamenteel zijn (zie bijvoorbeeld de paragrafen 7.5.6-7.5.9, waar de EXISTS wordt behandeld).

**7.5.1 Opzoeken en null waarden.** Veronderstel ter wille van het voorbeeld, dat leverancier L5 een status null in plaats van 30 heeft. Geef dan de nummers van de leveranciers met een status groter dan 25.

```
SELECT    LNO
FROM      L
WHERE     STATUS > 25
```

*Resultaat:*

```
LNO
---
L3
```

Leverancier L5 doet onder de gegeven aanname in het resultaat niet mee. Als een null waarde bij het evalueren van een conditie wordt vergeleken met een andere waarde, dan zal, ongeacht de vergelijkingsoperator, het resultaat van de vergelijking *nooit waar* zijn; zelfs niet als die andere waarde ook null is. Met andere woorden, als STATUS de waarde null heeft, dan zal geen van de volgende vergelijkingen tot *waar* evalueren:

```
STATUS > 25
STATUS <= 25
STATUS = 25
STATUS ~= 25
STATUS = NULL   (* Een syntactische fout, zie hierna *)
STATUS ~= NULL (* Hier geldt hetzelfde *)
```

In SQL wordt een speciale conditie aangeboden om de aan- of afwezigheid van null waarden te testen. De syntax van die conditie luidt:

```
kolomnaam  IS  [NOT]  NULL
```

Een voorbeeld van deze speciale conditie is het vragen naar de nummers van die leveranciers die een STATUS waarde gelijk aan null hebben.

```
SELECT    LNO
FROM      L
WHERE     STATUS IS NULL
```

*Resultaat:*

```
LNO
---
L5
```

De syntax "STATUS = NULL" is niet toegestaan, omdat *geen enkele kolomwaarde*, zelfs null zelf niet, geacht wordt gelijk te zijn aan null.

**7.5.2 Opzoeken met een subvraag.** Geef de namen van de leveranciers die artikel A2 kunnen leveren.

```
SELECT    LNAAM
FROM      L
WHERE     LNO IN
        ( SELECT    LNO
          FROM      LA
          WHERE     ANO = 'A2' )
```

*Resultaat:*

```
LNAAM
-----
Smit
Janse
Blaak
Klerk
```

Uitleg: Een subvraag is een SELECT-FROM-WHERE uitdrukking, die (eenvoudig geformuleerd) genest in een andere SELECT-FROM-WHERE uitdrukking voorkomt. Subvragen worden bij uitstek gebruikt om de verzameling waarden, waarin gezocht wordt via een "IN-conditie", te representeren (zie bovenstaand voorbeeld). Het systeem evalueert de totale vraag door eerst de geneste subvraag te te beantwoorden. Die subvraag geeft dan een verzameling nummers van die leveranciers die artikel A2 kunnen leveren. Dit is de verzameling ('L1','L2','L3','L4'). De oorspronkelijke vraag wordt nu gereduceerd tot de volgende eenvoudigere vraag:

```
SELECT    LNAAM
FROM      L
WHERE     LNO IN ('L1','L2','L3','L4')
```

De conditie in de WHERE-clausule van deze vraag wordt waar, dan en slechts dan als LNO een van de waarden L1, L2, L3 of L4 heeft. Dit geheel resulteert in het antwoord als hiervoor al gegeven.

Merk op dat het oorspronkelijke probleem ("Geef de namen van de leveranciers die artikel A2 leveren") evengoed met behulp van een join kan worden geformuleerd:

```
SELECT    L.LNAAM
FROM      L, LA
WHERE     L.LNO = LA.LNO
AND       LA.ANO = 'A2'
```

Uitleg: de equijoin van L en LA (over LNO) bestaat uit een tabel met 12 rijen (een voor elke rij in LA), waarin elke rij bestaat uit de LA rij uitgebreid met de LNAAM, STATUS en WNPL waarden van de leverancier geïdentificeerd door de LNO waarde. Van deze twaalf rijen zijn er vier van artikel A2. Het uiteindelijke resultaat wordt bereikt door de LNAAM waarden uit deze vier rijen te extraheren.

De twee formuleringen van de oorspronkelijke vraag (de één gebruik makend van een subvraag, de ander van een join) zijn beide correct. Het is zuiver een kwestie van smaak welke van de twee wordt geprefereerd (behalve dat, afhankelijk van de kwaliteit van het DBMS, de ene

formulering sneller afgehandeld zou kunnen worden dan de andere; in een ideaal systeem zou de gebruiker zich natuurlijk niet met zulke problemen moeten bezighouden).

**7.5.3 Subvraag met meerdere niveaus van nesting.** Geef de namen van de leveranciers die tenminste één rood artikel kunnen leveren.

```
SELECT   LNAAM
FROM     L
WHERE    LNO  IN
       ( SELECT   LNO
         FROM     LA
         WHERE    ANO  IN
                ( SELECT   ANO
                  FROM     A
                  WHERE    KLEUR = 'Rood' ) )
```

*Resultaat:*

```
LNAAM
-----
Smit
Janse
Klerk
```

Subvragen kunnen tot elk niveau worden genest. Opgave: geef een equivalente formulering voor deze vraag met behulp van een join.

**7.5.4 Subvraag met een andere vergelijkingsoperator dan IN.** Geef de nummers van die leveranciers die in dezelfde plaats gevestigd zijn als leverancier L1.

```
SELECT   LNO
FROM     L
WHERE    WNPL =
       ( SELECT   WNPL
         FROM     L
         WHERE    LNO = 'L1' )
```

*Resultaat:*

```
LNO
---
L1
L4
```

Als de gebruiker weet dat een bepaalde subvraag precies één waarde zal retourneren, dan mag een eenvoudige vergelijkingsoperator (zoals =, >, etc) worden gebruikt in plaats van de meer algemene IN.

**7.5.5 Functie in een subvraag.** Geef de nummers van die leveranciers die een statuswaarde kleiner dan de huidige maximum statuswaarde in de L tabel hebben.

```
SELECT    LNO
FROM      L
WHERE     STATUS <
        ( SELECT    MAX (STATUS)
          FROM      L )
```

*Resultaat:*

```
LNO
---
L1
L2
L4
```

**7.5.6 Vraag met EXISTS.** Geef de namen van de leveranciers die artikel A2 kunnen leveren (hetzelfde voorbeeld als in 7.5.2).

```
SELECT    LNAAM
FROM      L
WHERE     EXISTS
        ( SELECT    *
          FROM      LA
          WHERE     LA.LNO = L.LNO
          AND       LA.ANO = 'A2' )
```

Uitleg: EXISTS stelt hier de *existentiële kwantor* voor. De uitdrukking "EXISTS (SELECT * FROM ...)" wordt dan en slechts dan waar als het resultaat van de evaluatie van "SELECT * FROM ..." niet leeg is. Dit betekent voor het voorbeeld: dan en slechts dan als er een rij in LA is die aan de conditie in de EXISTS uitdrukking voldoet. Merk op dat de WHERE-conditie in de EXISTS uitdrukking naar de FROM tabel

op het buitenste niveau van de vraag verwijst. Dit gebeurt door middel van een expliciet gekwalificeerde kolomnaam (L.LNO in dit geval).

Beschouw, om te zien hoe het voorbeeld werkt, de ene LNAAM na de andere en kijk of de LNAAM de existentietest *waar* maakt. Veronderstel dat de eerste LNAAM 'Smit' is (zodat de corresponderende LNO L1 is). Is de verzameling LA rijen met LNO = L1 en ANO = A2 leeg of niet? Het antwoord is dat deze verzameling niet leeg is, zodat 'Smit' een van de namen is die in het antwoord moet voorkomen. Voor de andere LNAAM waarden wordt hetzelfde proces uitgevoerd. Sommige geven een lege verzameling, andere niet. De hele vraag kan als volgt worden verduidelijkt: "Bepaal de namen van die leveranciers die met artikel A2 verbonden zijn door het feit dat ze A2 kunnen leveren."

Ofschoon dit voorbeeld alleen maar aantoont dat er nog een formulering bestaat voor een vraag die we al konden formuleren via een subvraag of een join, is de EXISTS feitelijk een van de meest fundamentele en meest algemene taalelementen in de hele SQL taal. (Merk in het bijzonder op dat iedere vraag met een IN-conditie altijd geformuleerd kan worden met een EXISTS. Het omgekeerde is echter niet waar.) De ontkennende vorm NOT EXISTS is van bijzonder belang voor een zekere klasse van (tamelijk complexe) vragen; we geven daarom enige voorbeelden van die ontkennende vorm.

**7.5.7 Vraag met NOT EXISTS.** Geef de namen van de leveranciers die artikel A2 niet kunnen leveren.

```
SELECT   LNAAM
FROM     L
WHERE    NOT EXISTS
       ( SELECT   *
         FROM     LA
         WHERE    LA.LNO = L.LNO
         AND      LA.ANO = 'A2' )
```

*Resultaat:*

```
LNAAM
-----
Adams
```

Merk op dat deze vraag ook kan worden uitgedrukt door gebruik te maken van een ontkennende vorm van IN (zie de formulering hierna).

```
SELECT   LNAAM
FROM     L
WHERE    LNO NOT IN
       ( SELECT   LNO
         FROM     LA
         WHERE    LA.ANO = 'A2' )
```

**7.5.8 Vraag met NOT EXISTS.** Geef de namen van de leveranciers die alle artikelen kunnen leveren.
Deze vraag is niet te formuleren met behulp van NOT IN (*Vraag*: waarom niet?). Bovendien ontbreekt in SQL de universele kwantor ALL. In plaats daarvan zal iedere conditie waarin op natuurlijke wijze de ALL zou voorkomen, moeten worden geherformuleerd.

De formulering van 7.5.7 helpt ons op weg: daar worden de leveranciers bepaald die A2 niet kunnen leveren. In ons probleem moet de naam van een leverancier worden afgedrukt als voor *geen enkel* artikel geldt, dat deze leverancier dit artikel *niet* kan leveren.

```
SELECT   LNAAM       (* geef naam *)
FROM     L           (* van leveranciers L waarvoor *)
WHERE    NOT EXISTS                (* niet bestaat *)
       ( SELECT   *
         FROM     A                (* een artikel dat *)
         WHERE    NOT EXISTS       (* niet leverbaar is *)
                ( SELECT   *
                  FROM     LA
                  WHERE    LA.LNC = L.LNO   (* door L *)
                  AND      LA.ANO = A.ANO       ) )
```

*Resultaat:*

```
LNAAM
-----
Smit
```

**7.5.9 Vraag met NOT EXISTS.** Geef de nummers van de leveranciers die alle artikelen kunnen leveren die L2 ook kan leveren.

Een manier om dit tamelijk complexe probleem aan te pakken is het op te splitsen in een aantal eenvoudiger problemen en die één voor één te behandelen.

Zo kunnen we bijvoorbeeld eerst de nummers van de door leverancier L2 leverbare artikelen bepalen.

```
SELECT   ANO
FROM     LA
WHERE    LA.LNO = 'L2'
```

*Resultaat:*

```
ANO
---
A1
A2
```

Door gebruik te maken van SQL faciliteiten om tijdelijke tabellen te creëren, is het mogelijk om het resultaat in een tabel te bewaren. Noem deze tabel TEMP.

Daarna bepalen we de nummers van die leveranciers, die alle artikelen in TEMP kunnen leveren (dit lijkt sterk op de aanpak van 7.5.8).

```
SELECT   DISTINCT LNO
FROM     LA  LAX
WHERE    NOT EXISTS
       ( SELECT   *
         FROM     TEMP
         WHERE    NOT EXISTS
                ( SELECT   *
                  FROM     LA  LAY
                  WHERE    LAY.LNO = LAX.LNO
                  AND      LAY.ANO = TEMP.ANO ) )
```

*Resultaat:*

```
LNO
---
L1
L2
```

Merk op dat deze vraag wel afwijkt van die in de vorige paragraaf, omdat het nodig is om tenminste één alias te gebruiken. We extraheren immers LNO waarden uit de LA tabel in plaats van LNAAM waarden van de L tabel en dus moeten we in staat zijn om gelijktijdig twee verschillende verwijzingen naar de LA tabel te doen.

Hierna kan de tijdelijke tabel TEMP worden verwijderd.

Het is vaak het beste om complexe vragen op een stapsgewijze manier te behandelen. Het is echter ook mogelijk om de hele vraag in een SQL vraag te formuleren, waardoor de behoefte aan een tijdelijke tabel TEMP verdwijnt.

```
SELECT   DISTINCT LNO
FROM     LA  LAX
WHERE    NOT EXISTS
       ( SELECT   *
         FROM     LA  LAY
         WHERE    LNO = 'L2'
         AND      NOT EXISTS
                ( SELECT   *
                  FROM     LA  LAZ
                  WHERE    LAZ.LNO = LAX.LNO
                  AND      LAX.ANO = LAY.ANO ) )
```

## 7.6 BIJWERKOPDRACHTEN

SQL bevat een drietal bijwerkopdrachten, namelijk de UPDATE (voor het wijzigen), DELETE (voor het verwijderen) en INSERT (voor het invoegen). In hoofdstuk 5 zijn een aantal taalelementen ingevoerd om de integriteit van een database te verhogen (zie paragraaf 5.4). Nog maar weinig geïmplementeerde DBMS'en bieden de daar genoemde (of vergelijkbare) taalelementen aan. Dit geldt, helaas, ook voor de meeste systemen die SQL ondersteunen. Er wordt vanaf nu aangenomen, dat de entiteit- en de referentie-integriteitsregels door het DBMS worden gehandhaafd, maar dat de faciliteiten van paragraaf 5.4 nog niet worden aangeboden.

Voordat we een aantal voorbeelden geven, vermelden we eerst de syntax en de functie van de SQL bijwerkopdrachten.

*UPDATE*

```
  UPDATE   tabelnaam
  SET      { kolomnaam = { expressie | NULL } } ...
[ WHERE    conditie ]
```

Alle rijen in genoemde tabel die aan "conditie" voldoen, worden gewijzigd in overeenstemming met de toekenningen ("kolom = expressie") in de SET-clausule.

*DELETE*

```
  DELETE
  FROM     tabelnaam
[ WHERE    conditie ]
```

Alle rijen in "tabel" die aan "conditie" voldoen, worden verwijderd.

*INSERT*

```
INSERT
INTO     tabelnaam [ ( { kolomnaam }  ...) ]
VALUES   ( { constante | NULL } ...)
```

of

```
INSERT
INTO     tabelnaam [ ( { kolomnaam }  ...) ]
SELECT-opdracht
```

In de eerste vorm wordt een rij aan de genoemde tabel toegevoegd met de gespecificeerde waarden voor de gespecificeerde kolommen. (De i-de constante in de lijst van constanten correspondeert met de i-de kolom in de lijst van kolommen.)

In de tweede vorm wordt de SELECT-opdracht geëvalueerd en een kopie van het resultaat (in het algemeen vele rijen) wordt aan de genoemde tabel toegevoegd; de i-de kolom van het resultaat correspondeert met de i-de kolom in de lijst van kolommen. In beide gevallen staat het weglaten van de lijst met kolommen gelijk aan het specificeren van de lijst van alle kolommen in de tabel (in de van links-naar-rechts volgorde van de CREATE TABLE opdracht).

**7.6.1 Update van één rij.** Verander de kleur van artikel A2 in geel, verhoog het gewicht met 5 en maak de plaats gelijk aan "onbekend" (null).

```
UPDATE   A
SET      KLEUR = 'Geel',
         GEW = GEW + 5,
         PLAATS = NULL
WHERE    ANO = 'A2'
```

Voor elke rij die wordt gewijzigd (dat wil zeggen, elke rij die voldoet aan de WHERE-conditie, of alle rijen als er geen WHERE-clausule is), geldt dat in de SET-clausule voorkomende verwijzingen naar kolommen in die rij betrekking hebben op de waarden van die kolommen *voordat* een toekenning van de SET-clausule heeft plaatsgevonden.

**7.6.2 Update van een aantal rijen.** Verdubbel de status van alle Londense leveranciers.

```
UPDATE   L
SET      STATUS = STATUS * 2
WHERE    WNPL = 'Londen'
```

**7.6.3 UPDATE met een subvraag.** Maak de hoeveelheid van alle door Londense leveranciers leverbare artikelen gelijk aan 0.

```
UPDATE    LA
SET       HOEV = 0
WHERE     'Londen' =
        ( SELECT    WNPL
          FROM      L
          WHERE     L.LNO = LA.LNO )
```

**7.6.4 UPDATE van meerdere tabellen.** Verander het leverancier-nummer van L2 in L9.

```
UPDATE    L
SET       LNO = 'L9'
WHERE     LNO = 'L2'

UPDATE    LA
SET       LNO = 'L9'
WHERE     LNO = 'L2'
```

Het is niet mogelijk om in één UPDATE opdracht meer dan een tabel te wijzigen. Met andere woorden, de UPDATE-clausule moet *precies één tabel* specificeren. In het voorbeeld hebben we een integriteitsprobleem en wel met de referentie-integriteit: de database wordt na de eerste wijziging inconsistent, omdat zij leveranties bevat waarvoor geen corresponderende leverancier aanwezig is. De database blijft inconsistent totdat de tweede wijziging is uitgevoerd. (Het verwisselen van de wijzigingen helpt natuurlijk niet; zie ook de opmerking in het begin van deze paragraaf.)

**7.6.5 DELETE van één rij.** Verwijder leverancier L1.

```
DELETE
FROM      L
WHERE     LNO = 'L1'
```

Als in de tabel LA staat dat leverancier L1 bepaalde artikelen kan leveren, dan wordt door deze DELETE de database inconsistent; zie ook het vorige voorbeeld. (Net als de UPDATE kent ook de DELETE geen variant die op meer dan één tabel werkt.)

**7.6.6 DELETE van een aantal rijen.** Verwijder alle leveranciers uit Madrid.

```
DELETE
FROM     L
WHERE    WNPL = 'Madrid'
```

**7.6.7 DELETE van alle rijen.** Verwijder alle LA rijen.

```
DELETE
FROM     LA
```

Let wel, de tabel LA is hierna leeg, maar blijft wel bestaan. Er kunnen dus invoegingen in LA gedaan worden. Om een tabel te verwijderen, kent SQL een andere opdracht (DROP TABLE).

**7.6.8 DELETE met een subvraag.** Verwijder de LA rijen van de Londense leveranciers.

```
DELETE
FROM     LA
WHERE    'Londen' =
       ( SELECT   WNPL
         FROM     L
         WHERE    L.LNO = LA.LNO )
```

**7.6.9 INSERT van één rij.** Voeg artikel A7 (plaats Athene, gewicht 24, naam en kleur momenteel onbekend) toe aan tabel A.

```
INSERT
INTO     A ( ANO, PLAATS, GEW )
VALUES   ( 'A7', 'Athene', 24 )
```

Een nieuwe A-rij wordt gemaakt met het gespecificeerde artikelnummer, plaats en gewicht. Kleur en naam worden null gemaakt. De volgorde van de kolommen in de INSERT opdracht hoeft niet gelijk te zijn aan de volgorde van de kolommen in de SQL CREATE TABLE opdracht.

**7.6.10 INSERT van één rij met weggelaten kolomnamen.** Voeg tupel A8 (naam Tandrad, kleur Roze, gewicht 14, plaats Nice) toe aan A.

```
INSERT
INTO     A
VALUES   ('A8', 'Tandrad', 'Roze', 14, 'Nice' )
```

Het weglaten van de lijst van kolommen staat gelijk aan het specificeren van een lijst van alle kolommen van de tabel in de volgorde waarin ze in de CREATE TABLE opdracht ook werden genoemd. Omdat tabellen uitgebreid kunnen worden met kolommen, is deze manier van

specificeren niet van gevaren ontbloot. Zodra een gebruiker de nieuwe tabeldefinitie niet kent, kunnen er fouten ontstaan.

**7.6.11 INSERT van één rij.** Voeg nieuwe leveringinformatie toe met leveranciernummer L20, artikelnummer A20 en hoeveelheid 1000.

```
INSERT
INTO      LA ( LNO, ANO, HOEV )
VALUES    ('L20','A20',1000)
```

Net als bij de UPDATE en de DELETE kunnen ook bij de INSERT, tenzij er geschikte acties worden uitgevoerd, problemen optreden met betrekking tot de referentie-integriteit. Hier zal gekeken moeten worden of L20 en A20 in L respectievelijk A als primaire sleutels voorkomen.

**7.6.12 INSERT van een aantal rijen.** Neem voor ieder artikel in de LA tabel de naam en de totale hoeveelheid op in de database.

```
CREATE     TABLE TEMP
         ( ANO       CHAR (6),
           TOTHOEV   INTEGER )

INSERT
INTO       TEMP   (  ANO, TOTHOEV  )
           SELECT    ANO, SUM (HOEV)
           FROM      LA
           GROUP     BY ANO
```

De SELECT wordt uitgevoerd als een normale SELECT; het resultaat wordt echter niet aan de gebruiker gegeven, maar gekopieerd in de tabel TEMP. Nu kan de gebruiker met de kopie doen wat hij wil; hij kan TEMP afdrukken, bijwerken of gebruiken om andere resultaten af te leiden. Geen van deze operaties heeft enige invloed op de oorspronkelijke gegevens in de database. Uiteindelijk kan de gebruiker de tabel, indien gewenst, verwijderen met een:

```
DROP   TABLE TEMP
```

Het voorgaande voorbeeld toont zeer fraai aan waarom de geslotenheidseigenschap van relationele systemen zo belangrijk is. Het voorbeeld werkt goed, omdat het resultaat van een SELECT weer een andere tabel is. Het zou *niet* gewerkt hebben als het resultaat iets anders dan een tabel geweest zou zijn.

Het is niet nodig dat de tabel TEMP leeg is als er een INSERT van een aantal rijen op gaat plaatsvinden. In het voorbeeld was dat toevalligerwijs zo. Als TEMP niet leeg is, dan worden nieuwe rijen gewoon aan de reeds bestaande verzameling rijen toegevoegd.

## 7.7 CONCLUDERENDE OPMERKINGEN

We hebben nu alle kenmerken van SQL doorgenomen die we in dit boek behandelen. De volgende kenmerken zijn aan de orde geweest:

- de basis SELECT-clausule met gebruik van DISTINCT, constanten, expressies en "SELECT *"
- de FROM-clausule met het gebruik van aliassen
- de WHERE-clausule met daarbij
  - vergelijkingsoperatoren =,~=,>,>=,~>,<,<=,~<
  - join-condities
  - booleaanse operatoren AND,OR,NOT
- de ingebouwde functies COUNT, SUM, AVG, MAX, MIN en het gebruik van de ORDER BY-, GROUP BY- en HAVING-clausules
- het gebruik van subvragen en de [NOT] IN vergelijkingsoperator
- het gebruik van de existentiële kwantor EXISTS (en speciaal ook de ontkennende vorm NOT EXISTS)
- de bijwerkoperatoren UPDATE, DELETE en INSERT.

Het is zinvol om op te merken, dat het bestaan van *slechts vier* DML opdrachten een van de redenen is voor het naar verhouding eenvoudige karakter van de taal. Het feit dat er maar vier opdrachten behoeven te zijn, is een gevolg van de eenvoud van het relationele data model. Zoals al in hoofdstuk 3 is vermeld, kan informatie in het relationele model op slechts één manier worden weergegeven. Dit in tegenstelling tot bijvoorbeeld DBTG-systemen (zie de hoofdstukken 8-10), waar informatie in de vorm van records en verbindingen tussen records kan worden weergegeven. Dit houdt in dat voor elk van de basisfuncties (opzoeken, wijzigen, invoegen en verwijderen) meer dan één operatie moet worden aangeboden (zie ook hoofdstuk 10).

Tot slot presenteren we een zeer gekunsteld voorbeeld, dat laat zien hoe een aantal (maar lang niet alle) faciliteiten van SQL in een SELECT opdracht kan worden gebruikt. We geven ook, en dat is belangrijker, een *conceptueel* algoritme voor de afhandeling van die vraag. Dat wil dus niet zeggen dat de vraag op die manier door het DBMS afgehandeld *moet* worden; er zijn mogelijk (en zelfs waarschijnlijk) snellere afhandelingswijzen denkbaar. Merk op, dat in zekere zin de SELECT de belangrijkste van de vier SQL opdrachten is; de andere drie moeten immers impliciet of expliciet door een geschikte SELECT worden voorafgegaan. In het geval van een DELETE bijvoorbeeld, moet het systeem eerst een impliciete SELECT uitvoeren om de te verwijderen gegevens op te sporen.

*Voorbeeld.*
Bepaal alle rode en blauwe artikelen, waarvan de totale hoeveelheid groter is dan 350. Hierbij moeten die LA rijen worden uitgesloten, die een

hoeveelheid kleiner of gelijk aan 200 hebben. Geef van die artikelen het nummer, het gewicht in grammen, de kleur en de maximale hoeveelheid waarin dat artikel leverbaar is.

```
SELECT   A.ANO, 'Gewicht in gr. =', A.GEW * 454,
         A.KLEUR, 'Max geleverde hoev =',
         MAX (LA.HOEV)
FROM     A, LA
WHERE    A.ANO = LA.ANO
AND      ( A.KLEUR = 'Rood' OR A.KLEUR = 'Blauw' )
AND      LA.HOEV > 200
GROUP    BY A.ANO
HAVING   SUM (HOEV) > 350
```

*Resultaat:*

```
ANO                          KLEUR
-------------------------------------------------------------
A1  Gewicht in gr. = 5448  Rood   Max geleverde hoev. = 300
A5  Gewicht in gr. = 5448  Blauw  Max geleverde hoev. = 400
A3  Gewicht in gr. = 7718  Blauw  Max geleverde hoev. = 400
```

*Uitleg van de conceptuele afhandeling van deze vraag:*
De clausules van een SELECT worden toegepast in de volgorde die wordt gesuggereerd door de geschreven volgorde. Een uitzondering hierop is de SELECT-clausule zelf, die wordt uitgevoerd *na* de HAVING. We kunnen ons voorstellen dat het resultaat op de volgende wijze zou kunnen zijn verkregen.

1. *FROM.* De FROM-clausule wordt gebruikt om een nieuwe tabel te leveren die het Cartesisch produkt van de tabellen A en LA is.

2. *WHERE.* Het resultaat van Stap 1 wordt gereduceerd door alle rijen te verwijderen die niet aan de condities in de WHERE-clausule voldoen. In het voorbeeld worden dus de rijen verwijderd die niet aan de volgende conditie voldoen:

```
...      A.ANO = LA.ANO
AND    ( A.KLEUR ='Rood' OR A.KLEUR = 'Blauw' )
AND      LA.HOEV > 200
```

3. *GROUP BY.* Het resultaat van Stap 2 wordt gegroepeerd, met als criterium de waarden van de kolom die in de GROUP BY clausule wordt genoemd. In het voorbeeld is dat de kolom A.ANO.

4. *HAVING.* Groepen die niet voldoen aan de conditie

```
SUM (HOEV) > 350.
```

worden uit het resultaat van Stap 3 verwijderd.

5. *SELECT.* Iedere groep in het resultaat van Stap 4 genereert als volgt slechts *een* rij. Eerst worden artikelnummer, gewicht, kleur en maximum hoeveelheid uit een groep gehaald. (Aangezien een artikelnummer per groep uniek en een primaire sleutel in A is, hoort bij een groep slechts één gewicht, kleur, etc.) Vervolgens wordt het gewicht naar grammen geconverteerd. Ten slotte worden de twee constanten 'Gewicht in gr. =' en 'Max geleverde hoev =' op de juiste plekken ingevoegd.

Zoals al eerder is opgemerkt, is het zojuist beschreven algoritme een zuiver conceptueel algoritme. Het geleverde resultaat is correct en kan dienen als een definitie van de semantiek (betekenis) van de oorspronkelijke SELECT opdracht. Indien het op deze manier uitgevoerd wordt, zal echter de beschreven afhandeling van de vraag waarschijnlijk zeer inefficiënt zijn, zowel gemeten in benodigde opslagruimte als in benodigde processortijd. Het zou bijvoorbeeld zeer ongelukkig zijn als het Cartesisch produkt zou worden bepaald zoals in Stap 1 is gesuggereerd. Beschouwingen zoals deze vormen precies de reden waarom relationele systemen een zogenaamde optimalisator nodig hebben. De taak van zo'n optimalisator is om een procedure te vinden, die hetzelfde resultaat produceert als het definiërende algoritme, maar die veel efficiënter dan dat algoritme is in termen van òf geheugenruimte òf rekentijd òf beide.

Tenslotte, kijkend naar de vraag zelf: het is natuurlijk waar dat de SELECT opdracht tamelijk complex is, maar bedenk hoeveel werk er wordt gedaan. Een conventioneel programma om hetzelfde te bewerkstelligen, kan in een taal zoals COBOL gemakkelijk negen pagina's lang worden. De genoemde SELECT opdracht is daarentegen slechts negen regels lang. De hoeveelheid werk die nodig is om het COBOL programma operationeel te maken, is veel groter dan de hoeveelheid werk nodig voor de getoonde SQL versie. In de praktijk zullen de meeste vragen natuurlijk veel eenvoudiger zijn dan dit laatste voorbeeld.

## OPGAVEN

Alle vragen in dit hoofdstuk zijn gebaseerd op de leveranciers-artikelen-projecten database, waarvan een inhoud al in hoofdstuk 4 werd gegeven. Misschien vindt u het prettig om de vragen te interpreteren in termen van de daar geschetste database-inhoud. Het kan verhelderend

werken als de resultaten op vragen worden verkregen door met de hand een en ander na te rekenen. Voor het gemak herhalen we de tabel- en kolomnamen van deze database. Primaire sleutels zijn onderstreept en de vreemde sleutels gecursiveerd.

```
L     (LNO, LNAAM, STATUS, WNPL)
A     (ANO, ANAAM, KLEUR, GEW, PLAATS)
P     (PNO, PNAAM, STAD)
LAP   (LNO, ANO, PNO, HOEV)
```

De vragen zijn in een aantal groepen ondergebracht oplopend in moeilijkheidsgraad. Het is verstandig om met de eenvoudige vragen te beginnen.

Er is ook een viertal SQL opdrachten geformuleerd (de vragen 7.37 tot 7.40) waarvan de 'terugvertaling' naar het Nederlands wordt gevraagd.

**Eenvoudige Vragen**

**7.1** Geef de volledige gegevens van alle projecten.

**7.2** Geef de volledige gegevens van alle Londense projecten.

**7.3** Geef de nummers van de leveranciers die aan project P1 leveren. Verwijder hierbij de duplicaten.

**7.4** Geef alle LA rijen met een hoeveelheid tussen 300 en 750.

**7.5** Geef alle <KLEUR,PLAATS> paren zodanig dat geen duplicaten voorkomen.

**Joins**

**7.6** Geef alle <LNO,ANO,PNO> drietallen zodanig dat de leverancier, het artikel en het project in dezelfde plaats thuishoren (dat wil zeggen WNPL=PLAATS=STAD).

**7.7** Geef alle <LNO,ANO,PNO> drietallen zodanig dat de leverancier, het artikel en het project niet in dezelfde plaats thuishoren.

**7.8** Geef alle <LNO,ANO,PNO> drietallen zodanig dat er geen tweetal van het bijbehorende drietal <WNPL,PLAATS,STAD> gelijk is.

**7.9** Geef de ANO waarden van alle artikelen die door Londense leveranciers geleverd worden.

**7.10** Geef de ANO waarden van die artikelen die worden geleverd door een Londense leverancier aan een Londens project.

**7.11** Geef alle <WNPL,STAD> paren zodat de leverancier in WNPL levert aan een project in STAD.

**7.12** Geef de ANO waarden van die artikelen die aan een project worden geleverd door een leverancier uit dezelfde stad als het project.

**7.13** Geef de PNO waarden van die projecten waaraan geleverd wordt door tenminste één leverancier uit een andere stad.

**7.14** Geef alle paren artikelnummers <ANO,ANO> zodanig dat er tenminste één leverancier is die beide artikelen levert.

### Built-in Functies

**7.15** Geef het totaal aantal projecten waaraan door leverancier L1 geleverd wordt.

**7.16** Geef de totale hoeveelheid van artikel A1 die door leverancier L1 geleverd wordt.

**7.17** Geef voor ieder artikel dat aan een project geleverd wordt het artikelnummer ANO, het projectnummer PNO en de corresponderende totale hoeveelheid.

**7.18** Geef de artikelnummers ANO van de artikelen die aan een project geleverd worden in een gemiddelde hoeveelheid van meer dan 320.

### Diversen

**7.19** Geef alle LAP rijen met een niet-null hoeveelheid.

**7.20** Geef het aantal LAP rijen met een niet-null hoeveelheid.

### Subvragen

**7.21** Geef de namen van de projecten waaraan door leverancier L1 geleverd wordt.

**7.22** Geef de kleuren van de artikelen die door leverancier L1 geleverd worden.

**7.23** Geef de nummers van de artikelen die aan een Londens project worden geleverd.

**7.24** Geef de PNO waarden van die projecten die tenminste één artikel door leverancier L1 geleverd krijgen.

**7.25** Geef de LNO waarden van die leveranciers die tenminste één artikel leveren dat ook door tenminste één leverancier geleverd wordt die tenminste één rood artikel levert.

**7.26** Geef de LNO waarden van leveranciers met een status kleiner dan die van leverancier L1.

**7.27** Geef de PNO waarden van die projecten die gelokaliseerd zijn in de stad die vooraan staat in de alfabetische lijst van steden.

**7.28** Geef de PNO waarden van die projecten waaraan artikel A1 wordt geleverd in een gemiddelde hoeveelheid die groter is dan de grootste hoeveelheid die van een artikel aan project P1 wordt geleverd.

**7.29** Geef de LNO waarden van die leveranciers die aan een project artikel A1 leveren in een hoeveelheid die groter is dan de gemiddelde hoeveelheid van artikel A1 voor datzelfde project.

**EXISTS**

**7.30** Herhaal Vraag 7.23 en gebruik EXISTS in de oplossing.

**7.31** Herhaal Vraag 7.24 en gebruik EXISTS in de oplossing.

**7.32** Geef de PNO waarden van die projecten waaraan niet een rood artikel door een Londense leverancier geleverd wordt.

**7.33** Geef de PNO waarden van die projecten die volledig door leverancier L1 voorzien worden.

**7.34** Geef de ANO waarden van die artikelen die aan alle Londense projecten geleverd worden.

**7.35** Geef de LNO waarden van die leveranciers die een artikel aan alle projecten leveren.

**7.36** Geef de PNO waarden van die projecten waaraan tenminste alle door leverancier L1 leverbare artikelen geleverd worden.

Zet de onderstaande SQL formuleringen om in het Nederlands.

**7.37**
```
SELECT  DISTINCT PNO
FROM    LAP LAPX
WHERE   NOT EXISTS
      ( SELECT  *
        FROM    LAP LAPY
        WHERE   LAPY.PNO = LAPX.PNO
        AND     NOT EXISTS
              ( SELECT  *
                FROM    LAP LAPZ
                WHERE   LAPZ.ANO = LAPY.ANO
                AND     LAPZ.LNO = 'L1' ) )
```

**7.38**
```
SELECT  DISTINCT PNO
FROM    LAP LAPX
WHERE   NOT EXISTS
      ( SELECT  *
        FROM    LAP LAPY
        WHERE   EXISTS
              ( SELECT  *
                FROM    LAP LAPA
                WHERE   LAPA.LNO = 'L1'
                AND     LAPA.ANO = LAPY.ANO )
        AND     NOT EXISTS
              ( SELECT  *
                FROM    LAP LAPB
                WHERE   LAPB.LNO = 'L1'
                AND     LAPB.ANO = LAPY.ANO
                AND     LAPB.PNO = LAPX.PNO ) )
```

**7.39**
```
SELECT  DISTINCT PNO
FROM    LAP LAPX
WHERE   NOT EXISTS
      ( SELECT  *
        FROM    LAP LAPY
        WHERE   EXISTS
              ( SELECT  *
                FROM    LAP LAPA
                WHERE   LAPA.ANO = LAPY.ANO
                AND     LAPA.PNO = LAPX.PNO )
        AND     NOT EXISTS
              ( SELECT  *
                FROM    LAP LAPB
                WHERE   LAPB.LNO = 'L1'
                AND     LAPB.ANO = LAPY.ANO
                AND     LAPB.PNO = LAPX.PNO ) )
```

**7.40**
```
SELECT  DISTINCT PNO
FROM    LAP LAPX
WHERE   NOT EXISTS
      ( SELECT  *
        FROM    LAP LAPY
        WHERE   EXISTS
              ( SELECT  *
                FROM    LAP S
                WHERE   S.LNO = LAPY.LNO
                AND     S.ANO IN
                      ( SELECT  ANO
                        FROM    A
                        WHERE   KLEUR = 'Rood' )
                AND     NOT EXISTS
                      ( SELECT  *
                        FROM    LAP T
                        WHERE   T.LNO=LAPY.LNO
                        AND     T.PNO=LAPX.PNO )))
```

### Bijwerkoperaties

**7.41** Verander de kleur van alle rode artikelen in oranje.

**7.42** Verwijder alle projecten waaraan geen artikelen geleverd worden.

**7.43** Verwijder alle projecten uit Rome en doe dat ook voor alle leveranties voor die projecten.

**7.44** Voeg een nieuwe leverancier (L10) toe aan de tabel L. De naam en de woonplaats zijn respectievelijk 'Witte' en 'New York'; de status is nog onbekend.

**7.45** Construeer een tabel die artikelnummers van artikelen bevat die ofwel geleverd kunnen worden door een Londense leverancier dan wel aan een Londens project.

**7.46** Construeer een tabel die projectnummers van projecten bevat die òf in Londen gelokaliseerd zijn òf door een Londense leverancier bevoorraad kunnen worden.

# 8
# DBTG
# Databases

## 8.1 INLEIDING

De afkorting DBTG verwijst naar de Data Base Task Group van het CODASYL COBOL Committee (CC). CODASYL is de afkorting van de naam van een groep computergebruikers die belang hebben bij zekere afspraken (standaards) ten aanzien van computertalen. Het CC is de commissie die verantwoordelijk is voor de ontwikkeling van de programmeertaal COBOL. Regelmatig wordt door deze commissie COBOL aangepast aan de nieuwste ontwikkelingen op computergebied. Een van die aanpassingen betrof het definiëren van en manipuleren met databases. In 1971 kwam de genoemde groep uit met het zogenaamde DBTG rapport [CODA71] waarin drie verschillende talen werden voorgesteld:

- de schema data definitietaal, die vanaf nu de schema DDL (naar data definition language) wordt genoemd,
- een subschema data definitietaal, vanaf nu (om voor de hand liggende redenen) de subschema DDL te noemen, en
- een data manipulatietaal, vanaf nu DML (naar data manipulation language) genaamd.

De laatste twee talen kunnen worden gezien als uitbreidingen op COBOL en vallen als zodanig dan ook binnen de bevoegdheden van de CC. De schema DDL is daarentegen een taal om een database met een netwerkstructuur te beschrijven en staat min of meer los van COBOL; het DBTG "schema" is analoog aan het begrip conceptuele schema. De schema DDL is opgezet om aan de eisen van vele programmeertalen te voldoen en niet alleen aan die van COBOL. Dit resulteerde al snel in de vorming van een nieuwe CODASYL commissie, die de Data Description

Language Committee (DDLC) werd genoemd. De DDLC werd gevormd met een zelfde taakstelling voor de schema DDL als die van de CC voor COBOL. Dat wil zeggen, de DDLC werd verantwoordelijk voor de voortdurende ontwikkeling van de schema DDL. Dit resulteerde in de loop van de jaren in een drietal herzieningen van de schema DDL, namelijk die van 1973, 1978 en 1981; sommige met veel, andere met weinig wijzigingen ten opzichte van de vorige versie (de versie van 1981 is beschreven in [KLUW87]). Ofschoon de verantwoordelijkheid voor de schema DDL nu ergens anders ligt, worden database systemen die volgens de specificaties van de Data Base Task Group zijn gemaakt nog steeds met DBTG systemen aangeduid. Deze conventie zullen wij hier volgen.

In een DBTG systeem wordt onder het begrip "gebruiker" een toepassingsprogrammeur verstaan en niet een eindgebruiker. De subschema DDL is de taal om een externe visie vast te leggen (het DBTG "subschema" correspondeert met de ANSI/SPARC term "extern schema"). Aangezien de gebruiker de programmeur is, moet de syntax van de subschema DDL compatibel zijn met de syntax van de programmeertaal die als hosttaal fungeert. Zo moet ook de DML een syntax hebben die compatibel is met de hosttaal. De DML en de subschema DDL zoals door DBTG in 1971 ontwikkeld werden, waren bedoeld voor COBOL; men had de hoop dat voor andere talen, zoals PL/1 en FORTRAN, soortgelijke pogingen ondernomen zouden worden.

In paragraaf 8.2 wordt de architectuur van een DBTG database systeem uiteengezet. De paragrafen 8.3, 8.4 en 8.5 behandelen het begrip "DBTG set", de meest typerende karakteristiek van het DBTG data model. In paragraaf 8.3 laten we zien hoe sets kunnen worden gebruikt om hiërarchische structuren weer te geven. In paragraaf 8.4 wordt hetzelfde gedaan voor netwerkstrukturen. Paragraaf 8.5 beschrijft een speciaal soort set, namelijk de zogenaamde "singuliere" set (in het Engels singular set), en paragraaf 8.6 presenteert een volledig (conceptueel) schema van de DBTG versie van de leveranciers-artikelen database. In paragraaf 8.7 wordt het onderwerp "membership class" behandeld. De kans bestaat dat men na het lezen van hoofdstuk 10 nog eens naar deze paragraaf zal teruggrijpen; in dat hoofdstuk worden namelijk de DML opdrachten behandeld die met de membership class te maken hebben. In paragraaf 8.8 wordt het begrip SET SELECTION uitgelegd. Opnieuw zal de lezer het misschien wenselijk vinden om deze paragraaf na behandeling van hoofdstuk 10 nog eens te herlezen. Immers de SET SELECTION hangt samen met zekere aspecten van de DML. Tenslotte wordt in de paragrafen 8.9 en 8.10 op subschema's ingegaan.

## 8.2 ARCHITECTUUR

De architectuur van een DBTG systeem wordt in Fig. 8.1 getoond. De "conceptuele visie" (een ANSI/SPARC term) wordt op typeniveau vastgelegd in het *schema*. Dit bestaat in essentie uit de definities van de verschillende *records* van de database, de *velden* (in het Engels "data-items") die zij bevatten en de *sets* waarin ze participeren. Het DBTG set begrip wordt in de volgende paragraaf uitgelegd. (In het algemeen gesproken is de DBTG set het middel om in een DBTG systeem associaties van het een-op-veel type weer te geven.) De interne visie (weer een ANSI/SPARC term) wordt beschreven door het opslagschema (storage schema). Dit schema wordt geformuleerd in de zogenaamde data opslagbeschrijvingstaal (Data Storage Description Language of DSDL). In hoofdstuk 11 wordt nader op het interne niveau ingegaan.

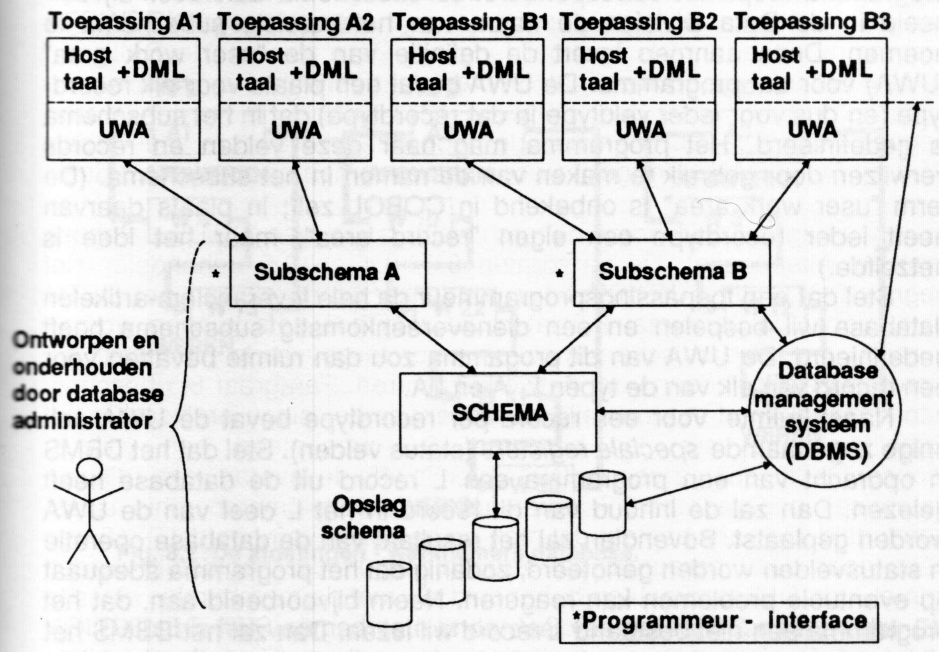



**Fig. 8.1 Architectuur van een DBTG systeem.**

De externe visie (een ANSI/SPARC term) wordt gedefinieerd in een *subschema*. Een subschema bestaat in essentie uit een specificatie van

dat deel van het schema waarin de gebruiker is geïnteresseerd, namelijk:
- de gewenste recordtypen,
- de gewenste veldtypen in die recordtypen, en
- de gewenste associatietypen (settypen) tussen deze recordtypen.

Alle record-, veld- en settypen die niet genoemd worden, zijn uitgesloten voor de gebruiker van het subschema. Het is niet mogelijk, tenminste niet in de momenteel beschikbare subschema DDL, om structuren in het subschema te definiëren, die niet al expliciet in het schema aanwezig zijn. Hierbij zou gedacht kunnen worden aan bijvoorbeeld een recordtype dat opgebouwd is uit velden van twee of meer andere recordtypen.

Tenslotte, zoals hiervoor al is uitgelegd, zijn de gebruikers van een DBTG database toepassingsprogrammeurs die programmeren in talen zoals COBOL, uitgebreid met database faciliteiten. Ieder toepassingsprogramma roept het corresponderende subschema aan door bijvoorbeeld in de Data Division de naam van het vereiste subschema te noemen. Deze aanroep levert de definitie van de "user work area" (UWA) voor dat programma. De UWA bevat een plaats voor elk recordtype (en dus voor ieder veldtype in dat recordtype) dat in het subschema is gedefinieerd. Het programma mag naar deze velden en records verwijzen door gebruik te maken van de namen in het subschema. (De term "user work area" is onbekend in COBOL zelf; in plaats daarvan heeft ieder recordtype een eigen "record area", maar het idee is hetzelfde.)

Stel dat een toepassingsprogrammeur de hele leveranciers-artikelen database wil bespelen en een dienovereenkomstig subschema heeft gedefinieerd. De UWA van dit progamma zou dan ruimte bevatten voor een record van elk van de typen L, A en LA.

Naast 'ruimte' voor een record per recordtype bevat de UWA ook enige zogenaamde *speciale registers* (status velden). Stel dat het DBMS in opdracht van een programma een L record uit de database heeft gelezen. Dan zal de inhoud van dit record in het L deel van de UWA worden geplaatst. Bovendien zal het resultaat van de database operatie in statusvelden worden genoteerd, zodanig dat het programma adequaat op eventuele problemen kan reageren. Neem bijvoorbeeld aan, dat het programma een niet bestaand L record wil lezen. Dan zal het DBMS het niet bestaan van dat record ontdekken en een foutcode in het juiste statusveld (DB-STATUS in dit geval) zetten. (Naast DB-STATUS worden door het DBMS na het afhandelen van bepaalde DML opdrachten nog andere speciale registers van een waarde voorzien.)

Samenvattend kan men stellen, dat de UWA de communicatiebuffer tussen programma en DBMS is (zie Fig. 8.1).

## 8.3 DE SETCONSTRUCTIE: HIERARCHISCHE VOORBEELDEN

**8.3.1 Hiërarchie met slechts één afhankelijk niveau.** We gaan uit van het reeds eerder beschreven voorbeeld betreffende afdelingen en werknemers. Er bestaat een *1:n associatietype* tussen de entiteittypen Werknemers en Afdelingen, dat aangeeft welke werknemers in welke afdeling werken.

In het DBTG data model worden *entiteiten* door *records* gerepresenteerd. Fig. 8.2 toont de database die gegevens over afdelingen en werknemers bevat. Deze database bevat twee recordtypen, namelijk AFD en WERKN die respectievelijk de entiteittypen Afdelingen en Werknemers representeren. *Kenmerken* van entiteiten worden door *velden* weergegeven (de DBTG term voor veld is 'data-item'). In het algemeen zal elk van deze recordtypen diverse veldtypen bevatten, ook al kent Fig. 8.2 slechts één veldtype per recordtype, namelijk ANO (afdelingnummer) voor AFD en WNO (werknemernummer) voor WERKN. Er zijn drie AFD records en negen WERKN records. Deze records zijn in drie sets van het type AFDWERKN gegroepeerd.

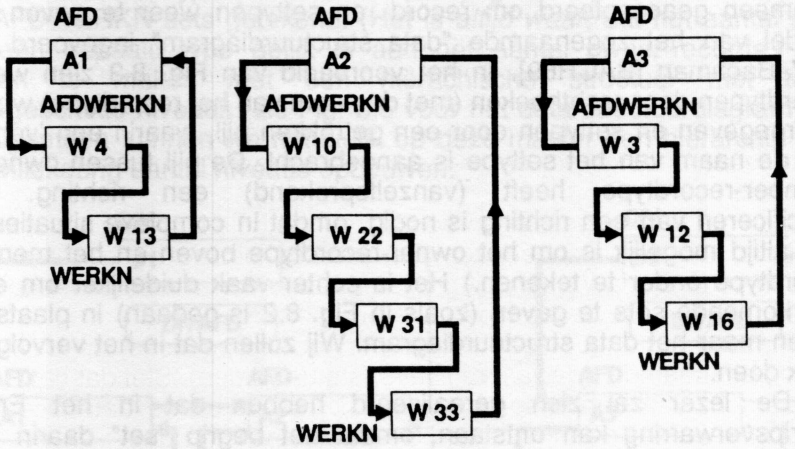



**Fig. 8.2 De afdelingen - werknemers database.**

De set is het voornaamste onderdeel van het DBTG data model. Elk settype heeft een recordtype als *owner* en een recordtype als *member*. In het voorbeeld zou het settype AFDWERKN in het schema gedeclareerd worden met AFD als owner en WERKN als member-recordtype. Elke set van een gegeven type bestaat uit precies één owner-record tezamen met nul of meer member-records. (Het 'nul geval' doet zich voor als bijvoorbeeld een bepaalde afdeling op een zeker moment geen werknemers heeft. In zo'n situatie spreken we toch van

een set, ook al heeft hij slechts het owner-record; de set wordt dan een lege set genoemd.)

In het voorbeeld bestaat er een 1:n associatietype tussen Afdelingen en Werknemers. Aangezien bij één owner nul, één of meer members kunnen horen, terwijl bij een member slechts een owner behoort, *representeert een settype een 1:n associatietype* tussen de verzameling owner-entiteiten en de verzameling member-entiteiten. Zo geeft het settype AFDWERKN een 1:n associatietype tussen Afdelingen en Werknemers weer.

De wijze waarop de owner-records (fysiek) verbonden zijn met de bijbehorende member-records is *niet* relevant voor de gebruiker. Een manier om deze verbindingen te realiseren (en het is bij lange na niet de enige) is via een wijzerketen, die start bij het owner-record, vervolgens alle member-records doorloopt om tenslotte naar de owner terug te keren (zoals in Fig. 8.2 is weergegeven). Ter wille van de eenvoud wordt deze methode vanaf nu gehanteerd; als in de database een andere methode zou worden gehanteerd, dan moet deze functioneel equivalent zijn aan de methode met de wijzerketens.

Het weergeven van sets met behulp van ketens van wijzers, zoals in Fig. 8.2 is gedaan, lijkt algemeen geaccepteerd te zijn. Evenzo is het algemeen geaccepteerd om record- en settypen weer te geven door middel van het zogenaamde "data structuurdiagram" ingevoerd door C.W. Bachman [BACH69]. In het voorbeeld van Fig. 8.3 zien we dat recordtypen door rechthoeken (met de naam van het recordtype) worden weergegeven en settypen door een getrokken pijl, waarin een 'venster' met de naam van het settype is aangebracht. De pijl tussen owner en member-recordtype heeft (vanzelfsprekend) een richting. (Het specificeren van een richting is nodig, omdat in complexe situaties het niet altijd mogelijk is om het owner-recordtype boven en het member-recordtype onder te tekenen.) Het is echter vaak duidelijker om enige voorkomende sets te geven (zoals in Fig. 8.2 is gedaan) in plaats van alleen maar het data structuurdiagram. Wij zullen dat in het vervolg ook vaak doen.

De lezer zal zich gerealiseerd hebben dat in het Engels begripsverwarring kan ontstaan, omdat het begrip "set" daarin twee betekenissen heeft, namelijk de betekenis van "set" in de DBTG zin en die van "set" (=verzameling) uit de wiskunde. De DBTG terminologie is ietwat ongelukkig, omdat men bijvoorbeeld wel eens over de 'set' van member-records (wiskundig bedoeld) behorende tot een bepaalde set (in de DBTG betekenis) wil spreken. In het Nederlands vermijden we vanaf nu dat probleem door af te spreken dat met "set" steeds de DBTG set en met "verzameling" de set uit de wiskunde wordt bedoeld.

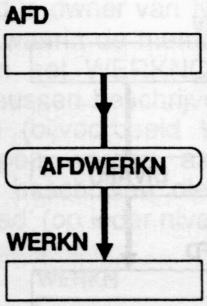



**Fig. 8.3 Data structuurdiagram van de afdelingen - werknemers database.**

**8.3.2 Hiërarchie met meer dan één ondergeschikt niveau.** Fig. 8.4 toont (een deel van) een database die gegevens over divisies, afdelingen en werknemers bevat. Dit voorbeeld illustreert dat een bepaald recordtype (in dit voorbeeld AFD) in het schema kan worden gedeclareerd als het member van een settype (DIVAFD) en de owner van een ander settype (AFDWERKN). In Fig. 8.4 zijn twee DIVAFD en drie AFDWERKN sets getekend. (Het is *altijd waar*, dat het aantal sets van een bepaald type gelijk is aan het aantal owner-records.) We hebben te maken met een hiërachische structuur met twee ondergeschikte niveaus (zie Fig. 8.5 voor het data structuurdiagram). In het algemeen kunnen we natuurlijk op deze manier een hiërarchie met een willekeurig aantal niveaus opbouwen.

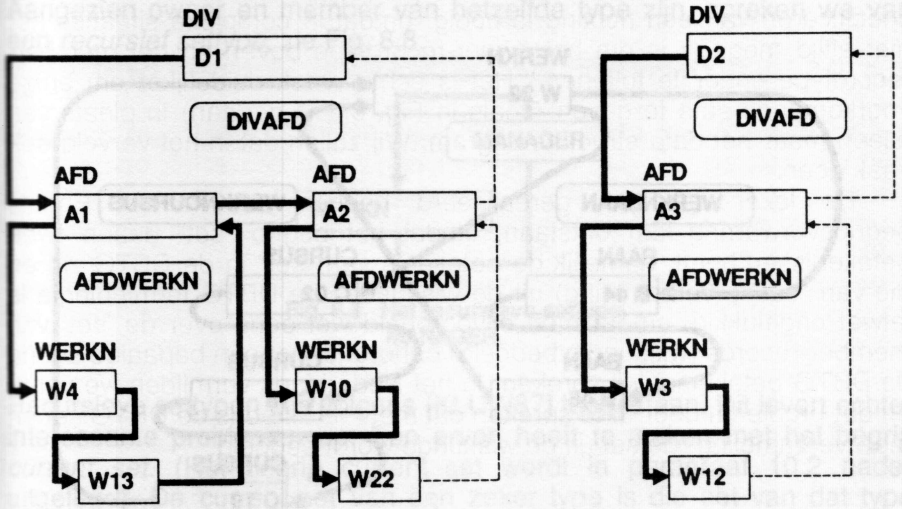



**Fig. 8.4 De divisies - afdelingen - werknemers database.**

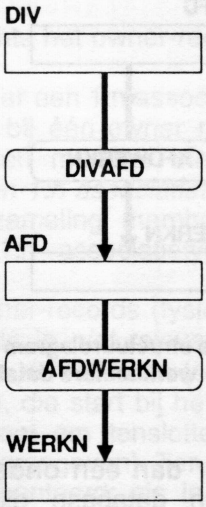



**Fig. 8.5 Data structuurdiagram van de divisies - afdelingen - werknemers database.**

**8.3.3 Hiërarchie met meer dan één recordtype op een ondergeschikt niveau.** Figuur 8.6 toont (een deel van) een database die gegevens over werknemers met door hen vervulde banen en door hen gevolgde cursussen bevat.

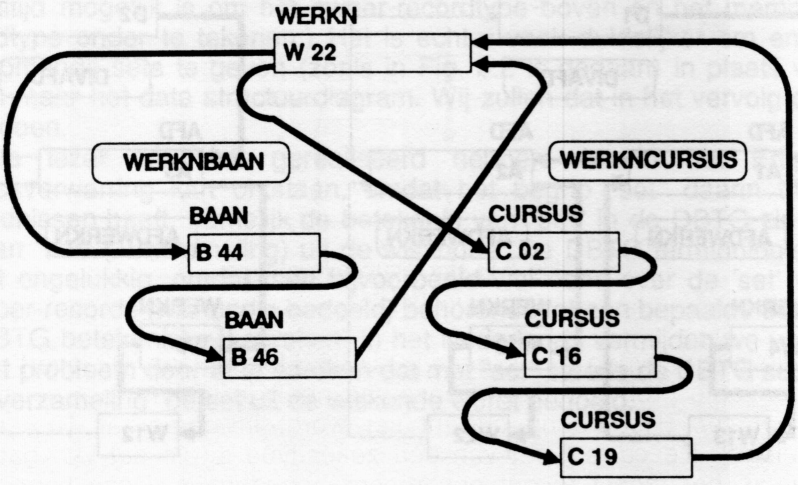



**Fig. 8.6 De werknemer - historie database.**

Ieder WERKN record is owner van twee sets, namelijk van een set van type WERKNBAAN, waarin de member-records de vervulde banen beschrijven, en van een set WERKNCURSUS, waarin de member-records de gevolgde cursussen beschrijven. In het algemeen kan in het schema een recordtype (bijvoorbeeld WERKN) als owner van een willekeurig aantal settypen worden aangewezen. We kunnen dus hiërarchische structuren beschrijven die niet alleen willekeurig diep, maar ook willekeurig 'breed' (op ieder niveau) zijn. Zie Fig. 8.7.

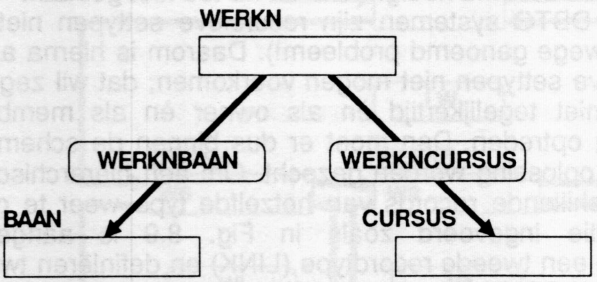



**Fig. 8.7 Hiërarchie met twee recordtypen op ondergeschikt niveau.**

**8.3.4 Hiërarchie met hetzelfde recordtype op meer dan één niveau.** Beschouw de database van de werknemers in een bedrijf. Iedere werknemer heeft een manager toegewezen gekregen en deze associatie relateert records van het type WERKN aan (andere) records van ditzelfde type. Deze associatie is 1:n, omdat bij iedere manager een aantal ondergeschikten hoort en bij een ondergeschikte slechts één manager. Dit is in DBTG uitstekend via een settype weer te geven. Aangezien owner en member van hetzelfde type zijn, spreken we van een *recursief settype*, zie Fig. 8.8.

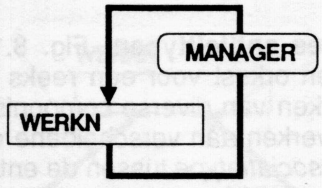



**Fig. 8.8 Het recursieve settype MANAGER.**

Recursieve settypen zijn volgens [KLUW87] toegestaan. Dit levert echter interessante problemen op. Eén ervan heeft te maken met het begrip *current set*. (Het begrip current set wordt in paragraaf 10.2 nader uitgelegd). De current set van een zeker type is die set van dat type waarvan het meest recent een record benaderd is. Deze current set speelt een rol in sommige DML opdrachten. Bij een recursief settype is

het begrip 'current set' echter niet altijd eenduidig te bepalen. Beschouw bijvoorbeeld het settype MANAGER. Stel dat het record van een werknemer die èn ondergeschikte èn manager is, wordt benaderd. Dan bepaalt dit record twee sets van type MANAGER, namelijk de set waarin hij ondergeschikte (member) is en de set waarin hij als manager (owner) genoteerd staat. Echter, de huidige DML (zie hoofdstuk 10) eist dat bij een settype slechts één current set hoort. Om de keuze te maken, is een uitbreiding van de DML nodig. Dit is tot nu toe niet gedaan.

In veel DBTG systemen zijn recursieve settypen niet toegestaan (juist al vanwege genoemd probleem!). Daarom is hierna aangenomen, dat recursieve settypen niet mogen voorkomen; dat wil zeggen dat een recordtype niet tegelijkertijd èn als owner èn als member van een settype mag optreden. Dan moet er dus binnen de schema DDL naar een andere oplossing worden gezocht. Om een hiërarchische structuur tussen verschillende records van hetzelfde type weer te geven, wordt een indirectie ingevoerd zoals in Fig. 8.9 is aangegeven. We introduceren een tweede recordtype (LINK) en definiëren twee settypen: WL (owner WERKN, member LINK) en LW (owner LINK, member WERKN). Elk WERKN record dat een manager representeert, is de owner van een WL set die precies een LINK record bevat. Dit LINK record is op zijn beurt de owner van een LW set die members van het type WERKN heeft. Deze WERKN records representeren de directe ondergeschikten van genoemde manager. Zo is bijvoorbeeld werknemer W1 de manager van de werknemers W6, W8,...; op zijn beurt is W6 de manager van W18,...; etc.

Fig. 8.10 geeft het data structuurdiagram. Merk op dat LINK records geen velden hoeven te bevatten.

## 8.4 DE SETCONSTRUCTIE: NETWERK VOORBEELDEN

**8.4.1 Netwerk met twee entiteittypen.** Fig. 8.11 geeft een database met de plannen van een orkest voor een reeks concertuitvoeringen. In elk concert worden werken van diverse componisten uitgevoerd en elke componist zal via zijn werken aan verscheidene concerten bijdragen. Er bestaat dus een *n:m associatietype* tussen de entiteittypen Concerten en Componisten. In DBTG mag binnen een gegeven settype een member-record slechts tot één set van dat type behoren. Het mag wel voorkomen, dat een member-record op verschillende momenten tot verschillende sets van een zeker type behoort. Om deze laatste twee punten te illustreren: het mag niet gebeuren dat een WERKN record (zie Fig. 8.2) op een zeker moment tot meer dan één AFDWERKN set behoort; wel kan een WERKN record op verschillende tijdstippen tot verschillende AFDWERKN sets behoren. Dit laatste betekent in het object systeem dat de werknemer achtereenvolgens in verschillende afdelingen heeft gewerkt.

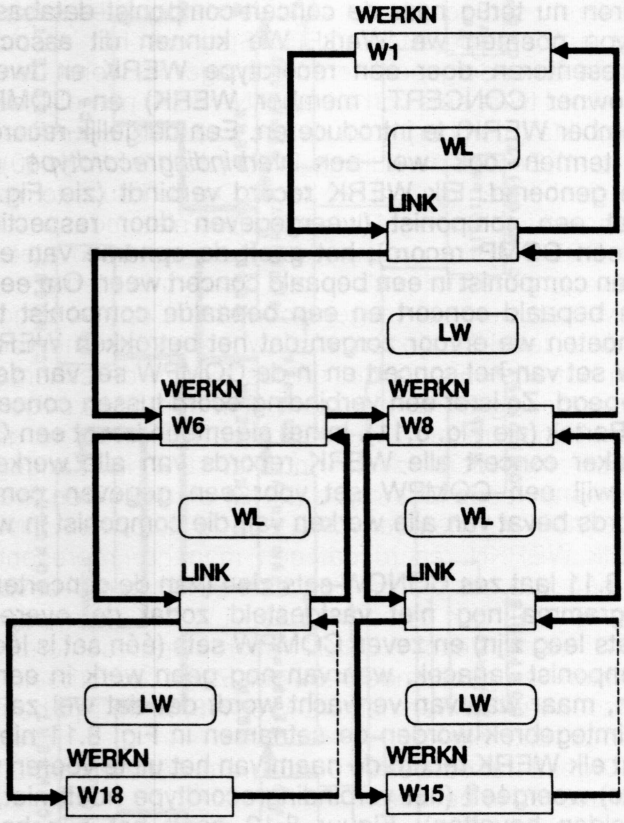



Fig. 8.9 De "management" database.

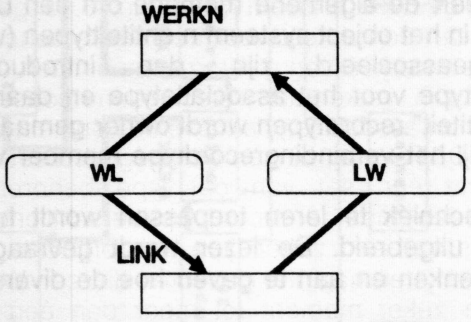



Fig. 8.10 Data structuurdiagram van de "management" database.

We keren nu terug naar de concert-componist database. Het n:m associatietype noemen we 'Werk'. We kunnen dit associatietype in DBTG representeren door een recordtype WERK en twee settypen CONCW (owner CONCERT, member WERK) en COMPW (owner COMP, member WERK) te introduceren. Een dergelijk recordtype wordt in DBTG termen ook wel een *verbindingrecordtype* (connection recordtype) genoemd. Elk WERK record verbindt (zie Fig. 8.11) een concert met een componist (weergegeven door respectievelijk een CONC en een COMP record); het geeft de opname van een bepaald werk van een componist in een bepaald concert weer. Om een koppeling tussen een bepaald concert en een bepaalde componist tot stand te brengen, moeten we ervoor zorgen dat het betrokken WERK record in de CONCW set van het concert en in de COMPW set van de componist wordt ingevoegd. Zo is er een verbindingrecord tussen concert C1 en de componist Bartok (zie Fig. 8.11). In het algemeen bevat een CONCW set van een zeker concert alle WERK records van alle werken voor dat concert, terwijl een COMPW set voor een gegeven componist alle WERK records bevat van alle werken van die componist in welk concert dan ook.

Figuur 8.11 laat zes CONCW sets zien (van de concerten C2 en C4 is het programma nog niet vastgesteld zodat de overeenkomstige CONCW sets leeg zijn) en zeven COMPW sets (één set is leeg, namelijk die van componist Janacek, waarvan nog geen werk in een concert is opgenomen, maar waarvan verwacht wordt dat dat wel zal gebeuren). Wegens ruimtegebrek worden de setnamen in Fig. 8.11 niet genoemd. Merk op dat elk WERK record de naam van het uit te voeren werk (Violin Conc. 2, etc) weergeeft (het verbindingrecordtype hoeft niet, maar mag dus wel velden bevatten). Figuur 8.12 geeft het bijbehorende data structuurdiagram.

**8.4.2 Netwerken met meer dan twee entiteittypen.** Het voorgaande voorbeeld illustreert de algemene methode om een DBTG database te construeren. Als in het object systeem n entiteittypen (voorgesteld door n recordtypen) geassocieerd zijn, dan introduceren we een verbindingrecordtype voor het associatietype en daarnaast n settypen. Elk van de n "entiteit" recordtypen wordt owner gemaakt van één van de n settypen, terwijl het verbindingrecordtype member wordt van elk van de n settypen.

Om deze techniek te leren toepassen wordt het voorbeeld van paragraaf 8.4.1 uitgebreid. De lezer wordt gevraagd om database-inhouden te bedenken en aan te geven hoe de diverse sets eruit gaan zien.

- Een nieuw entiteittype, SOLIST, wordt ingevoerd. Een solist kan op vele concerten optreden en een concert kan vele solisten hebben.

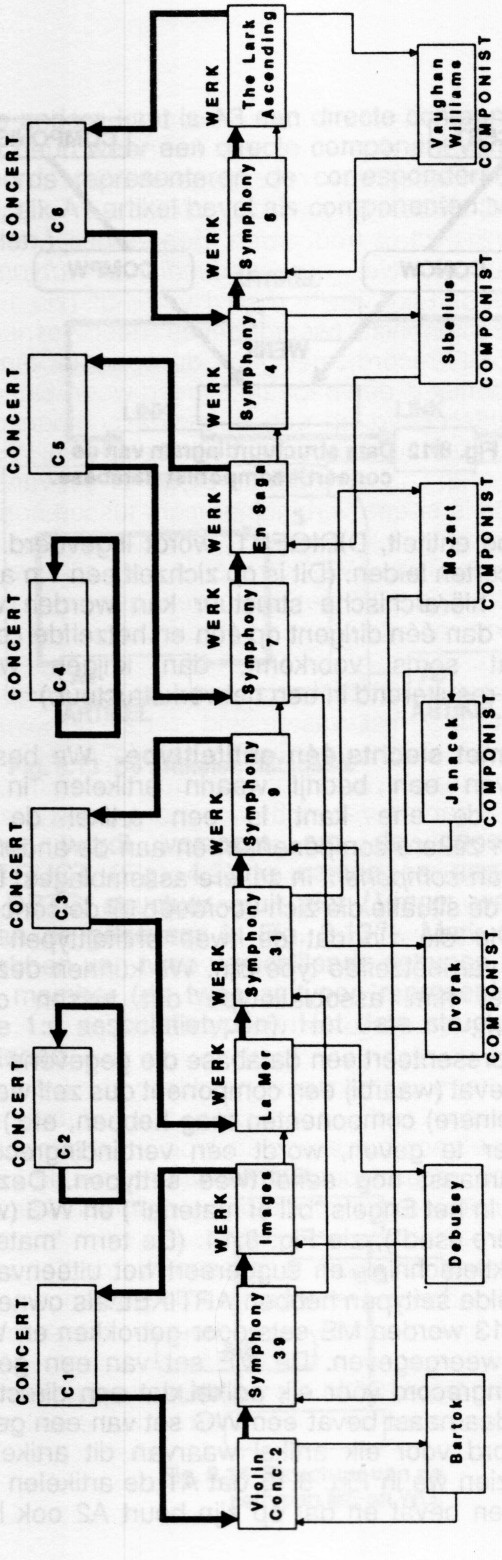



Fig. 8.11 De concert - componist database.

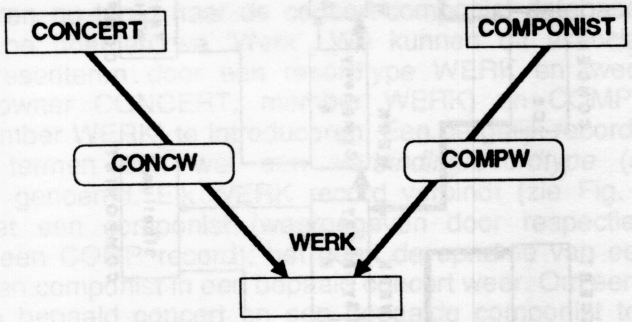



**Fig. 8.12 Data structuurdiagram van de concert - componist database.**

- Een nieuw type entiteit, DIRIGENT, wordt ingevoerd. Elke dirigent mag vele concerten leiden. (Dit is op zichzelf een 1:n associatietype, dat door een hiërarchische structuur kan worden weergegeven, maar als meer dan één dirigent op één en hetzelfde concert kunnen optreden, wat soms voorkomt, dan krijgen we een n:m associatietype resulterend in een netwerkstructuur.)

**8.4.3 Netwerken met slechts één entiteittype.** We beschouwen het object systeem van een bedrijf waarin artikelen in twee rollen voorkomen. Aan de ene kant is een artikel de *assemblage* (samenstelling) van zekere componenten en aan de andere kant speelt een artikel de rol van *component* in zekere assemblages. Dit is dus een speciaal geval van de situatie die zich voordeed bij de concert-componist database en wel in die zin dat de twee entiteittypen nu niet van verschillend, maar van hetzelfde type zijn. We kunnen dezelfde aanpak volgen als bij het n:m associatietype dat tussen concerten en componisten bestaat.

Figuur 8.13 representeert een database die gegevens over artikelen en componenten bevat (waarbij een component dus zelf weer een artikel is en zelf weer (kleinere) componenten mag hebben, etc.). Om het n:m associatietype weer te geven, wordt een verbindingrecordtype LINK ingevoerd en daarnaast nog eens twee settypen. Deze heten ME (materiaalexplosie, in het Engels "bill of material") en WG (waar gebruikt, in het Engels "where used"), zie Fig. 8.14. (De term 'materiaalexplosie' komt uit de produktietechniek en suggereert het uiteenvallen van een artikel in delen.) Beide settypen hebben ARTIKEL als owner en LINK als member. In Fig. 8.13 worden ME sets door getrokken en WG sets door gestippelde lijnen weergegeven. De ME set van een gegeven artikel bevat een verbindingrecord voor elk artikel dat een directe component van dat artikel is; daarnaast bevat een WG set van een gegeven artikel een verbindingrecord voor elk artikel waarvan dit artikel een directe component is. Zo zien we in Fig. 8.13 dat A1 de artikelen A2 en A3 als directe componenten bevat en dat op zijn beurt A2 ook het artikel A3

bevat; aan de andere kant is A3 een directe component van A1 en A2, en A2 op zijn beurt weer een directe component van A1. De getallen in de LINK records representeren de corresponderende hoeveelheden. (Bijvoorbeeld, elk A1 artikel bevat als componenten twee A2 artikelen en vier A3 artikelen.)

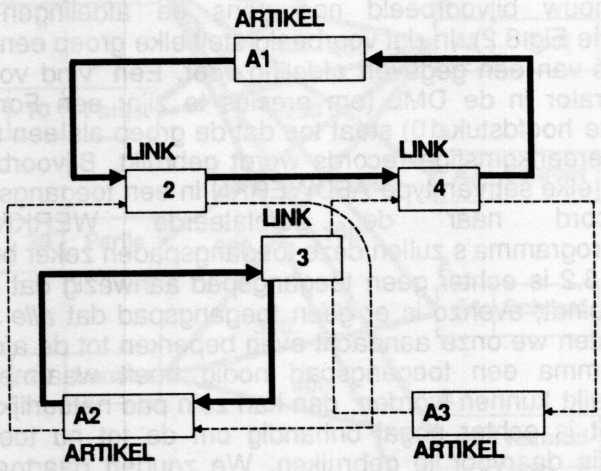



**Fig. 8.13 De artikelen database.**

De lezer wordt verzocht om de gegevens van relatie ARTIKEL_STRUCT (Fig. 4.3) te nemen en aan te geven hoe de equivalente DBTG structuur eruit ziet (*Vraag*: wat is de relationele weergave van de gegevens in Fig. 8.13?). Merk op dat we hier een voorbeeld hebben van twee verschillende settypen met dezelfde owner en dezelfde member (de twee settypen representeren natuurlijk twee verschillende 1:n associatietypen). Het data structuurdiagram wordt in Fig. 8.14 getoond.

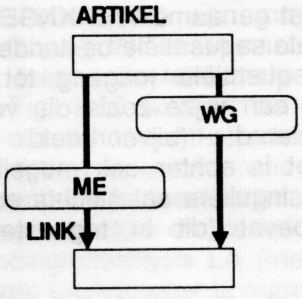



**Fig. 8.14 Structuur van de settypen ME en WG.**

## 8.5 SINGULIERE SETS

De setconstructie die in de twee voorafgaande paragrafen is ingevoerd, voorziet in een manier om member-records op een zodanige wijze te groeperen dat de records in een groep logisch met elkaar verbonden zijn. Beschouw bijvoorbeeld nog eens de afdelingen-werknemers database (zie Fig.8.2). In dat voorbeeld stelt elke groep een verzameling werknemers van een gegeven afdeling voor. Een "vind volgende in de groep" operator in de DML (om precies te zijn: een Format 3 FIND opdracht, zie hoofdstuk 10) staat toe dat de groep als een toegangspad naar de overeenkomstige records wordt gebruikt. Bijvoorbeeld, in Fig. 8.2 voorziet elke set van type AFDWERKN in een toegangspad van een afdelingsrecord naar de gerelateerde WERKN records; gebruikersprogramma's zullen deze toegangspaden zeker benutten.

In Fig. 8.2 is echter geen toegangspad aanwezig dat *alle* WERKN records verbindt; evenzo is er geen toegangspad dat *alle* AFD records relateert. Laten we onze aandacht even beperken tot de afdelingen. Als een programma een toegangspad nodig heeft waarmee alle AFD records bereikt kunnen worden, dan kan zo'n pad natuurlijk toegevoegd worden. Het is echter nogal onhandig om de tot nu toe ingevoerde setconstructie daarvoor te gebruiken. We zouden daartoe een nieuw recordtype, zeg AFD-OWNER, moeten invoeren. Dit nieuwe recordtype zou dan owner van een settype, zeg AFDSET, gemaakt moeten worden met AFD records als member-records. Er zou precies één record van type AFD-OWNER bestaan met mogelijk geen velden en er zou precies één set van type AFDSET zijn. Elk AFD record zou member van die set moeten zijn.

*Singuliere sets* bieden een gemakkelijkere oplossing voor het probleem. Een singuliere set mag gezien worden als een set die precies één exemplaar heeft en geen owner; dat wil zeggen, in het schema moet OWNER IS SYSTEM worden gespecificeerd in plaats van een of ander door de gebruiker bedacht recordtype. In het voorbeeld kunnen we daarom alle AFD records verzamelen in een singuliere set AFDSET en behoeven we niet het (gekunstelde) recordtype AFD-OWNER te introduceren. Op dezelfde wijze verzamelen we alle WERKN records in een andere singuliere set genaamd WERKNSET. Deze twee singuliere sets lijken erg op normale sequentiële bestanden. WERKNSET voorziet bijvoorbeeld in een sequentiële toegang tot alle WERKN records eventueel geordend op een wijze zoals die voor WERKNSET in het schema is gedefinieerd (bijvoorbeeld een ordening op werknemernummer). Het is echter ook mogelijk, ofschoon misschien ongebruikelijk, dat een singuliere set slechts een deelverzameling van alle member-records bevat (dit in tegenstelling tot een normale sequentieel bestand).

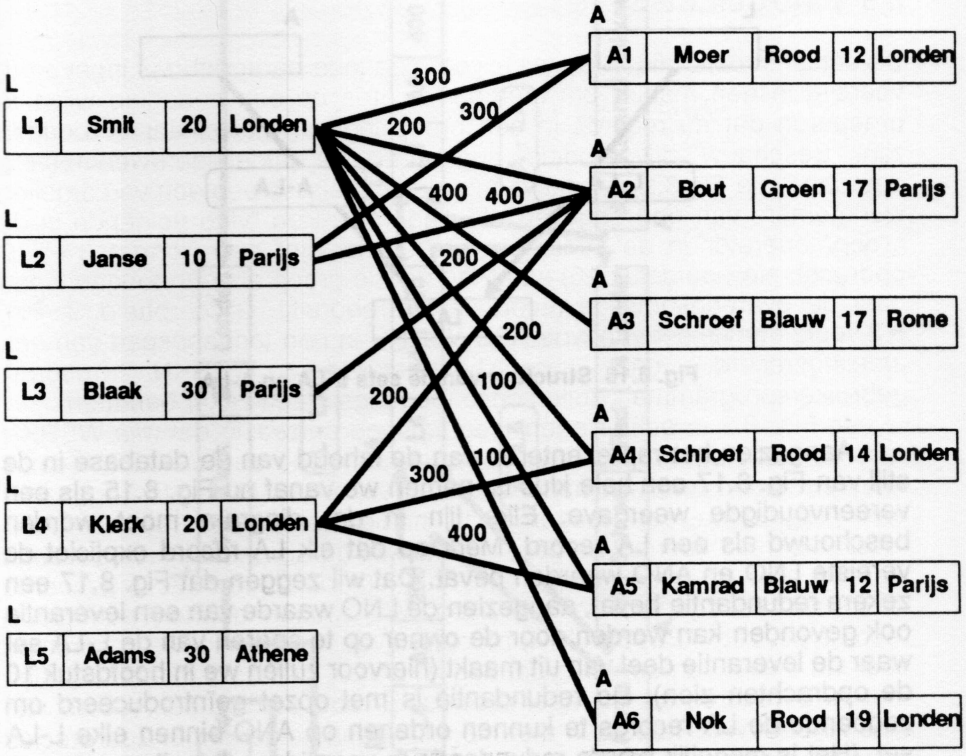



**Fig. 8.15 Voorbeeld gegevens (leveranciers en artikelen).**

## 8.6 EEN VOORBEELD SCHEMA

In deze paragraaf presenteren we een mogelijk schema voor de leveranciers-artikelen database. We keren naar dat voorbeeld terug om een directe vergelijking tussen de relationele en de netwerk aanpak mogelijk te maken. In Fig. 8.15 worden de gegevens van Fig. 7.1 opnieuw weergegeven om de netwerkstructuur te onderstrepen. De lijnen geven leverancier-artikel associaties weer (LA tupels in relationele terminologie) en de nummers de hoeveelheden.

We kiezen een netwerkstructuur om deze informatie weer te geven, introduceren een verbindingrecordtype LA (met velden LNO, ANO en HOEV) met twee settypen: L-LA (owner L, member LA) en A-LA (owner A, member LA). Een data structuurdiagram voor dit netwerk is in Fig. 8.16 gegeven; een deel van de actuele database is in Fig. 8.17 te zien.

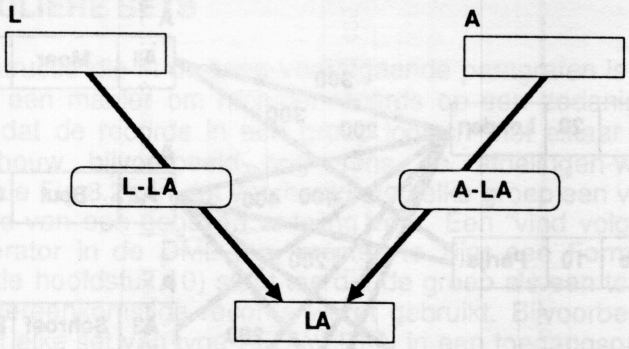



**Fig. 8.16 Structuur van de sets L-LA en A-LA.**

Aangezien het representeren van de inhoud van de database in de stijl van Fig. 8.17 een hele klus is, nemen we vanaf nu Fig. 8.15 als een vereenvoudigde weergave. Elke lijn in dat diagram moet worden beschouwd als een LA record. Merk op dat elk LA record expliciet de vereiste LNO en ANO waarden bevat. Dat wil zeggen dat Fig. 8.17 een zekere redundantie bevat, aangezien de LNO waarde van een leverantie ook gevonden kan worden door de owner op te sporen van de L-LA set waar de leverantie deel van uit maakt (hiervoor zullen we in hoofdstuk 10 de opdrachten zien). De redundantie is met opzet geïntroduceerd om zodoende de LA records te kunnen ordenen op ANO binnen elke L-LA set. (Het is mogelijk om de redundantie te vermijden, dat wil zeggen om ANO weg te laten, maar dan is het niet mogelijk om het DBMS automatisch de genoemde ordening te laten handhaven.)

Het schema is in Fig. 8.18 beschreven. In Appendix B zijn die delen van de DDL beschreven, die in dit hoofdstuk aan de orde zullen komen. Voor de volledige DDL, zie [KLUW87].

## Uitleg

Regel 1 kent een naam toe aan het schema. Deze regel beschrijft de zogenaamde schema entry.

Regel 2 definieert het bestaan van een recordtype L. Een recordtype wordt beschreven in een zogenaamde 'record entry' van het schema. Voor recordtype L omvat dat de regels 2 tot en met 8.

Regels 3 en 4 specificeren dat LNO IN L een sleutel is die L-SLT heet (de regels 2-4 vormen tezamen de record subentry). Deze naam L-SLT speelt verderop bij de SET SELECTION weer een rol (regels 31 en 32). Sleutels worden gebruikt om records te lokaliseren. Om bijvoorbeeld het L record van leverancier L1 te lokaliseren, kan de programmeur de juiste waarde (dat wil zeggen de waarde "L1") in het veld LNO IN L van de UWA plaatsen en dan de geschikte DML FIND opdracht geven (MOVE is de toekenningsopdracht in COBOL):

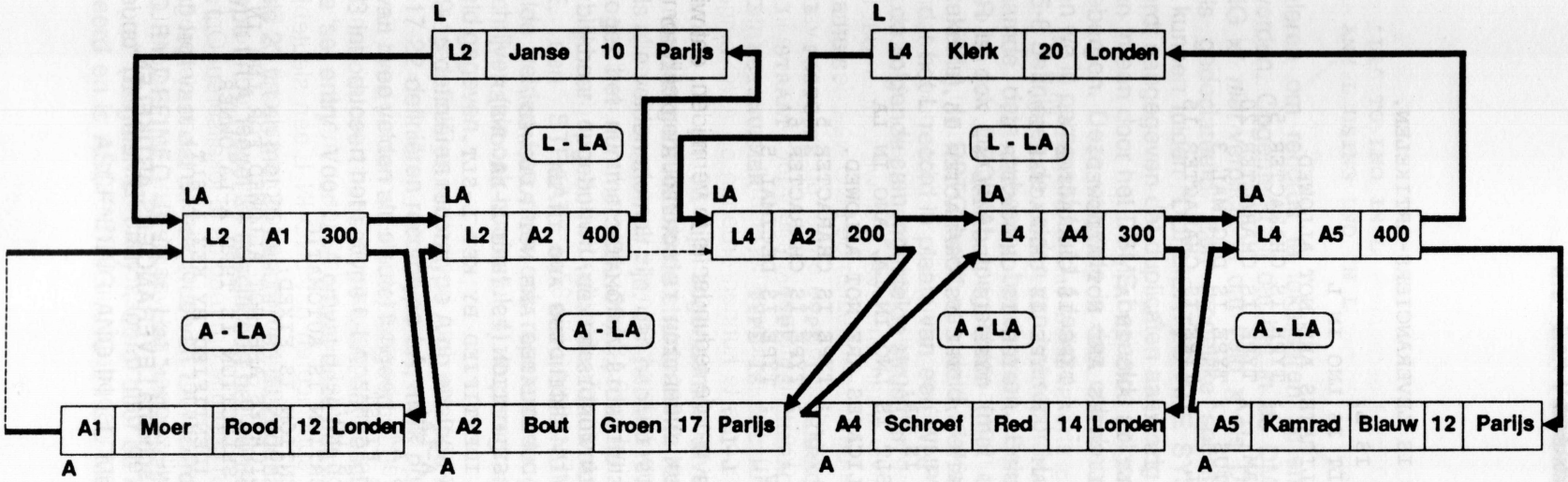



Fig. 8.17 (Deel van) de leveranciers - artikelen database.

```
1   SCHEMA NAME IS LEVERANCIERS-ARTIKELEN.

2   RECORD NAME IS L;
3       KEY L-SLT IS LNO IN L
4           DUPLICATES ARE NOT ALLOWED.
5       02  LNO      ; TYPE IS CHARACTER 5.
6       02  LNAAM    ; TYPE IS CHARACTER 20.
7       02  STATUS   ; TYPE IS DECIMAL 3.
8       02  WNPL     ; TYPE IS CHARACTER 15.

9   RECORD NAME IS A;
10      KEY A-SLT IS ANO IN A
11          DUPLICATES ARE NOT ALLOWED.
12      02  ANO      ; TYPE IS CHARACTER 6.
13      02  ANAAM    ; TYPE IS CHARACTER 20.
14      02  KLEUR    ; TYPE IS CHARACTER 6.
15      02  GEW      ; TYPE IS DECIMAL 6.
16      02  PLAATS   ; TYPE IS CHARACTER 15.

17  RECORD NAME IS LA;
18      KEY LA-SLT IS LNO IN LA, ANO IN LA
19          DUPLICATES ARE NOT ALLOWED.
20      02  LNO      ; TYPE IS CHARACTER 5.
21      02  ANO      ; TYPE IS CHARACTER 6.
22      02  HOEV     ; TYPE IS DECIMAL 5.

23  SET NAME IS L-LA;
24      OWNER IS L;
25      ORDER FOR INSERTION IS SORTED BY DEFINED KEYS.
26      MEMBER IS LA;
27          INSERTION IS AUTOMATIC
28          RETENTION IS FIXED;
29          KEY IS ASCENDING ANO IN LA
30              DUPLICATES ARE NOT ALLOWED
31          SET SELECTION IS THRU L-LA OWNER
32              IDENTIFIED BY KEY L-SLT.

33  SET NAME IS A-LA;
34      OWNER IS A;
35      ORDER IS FIRST.
36      MEMBER IS LA;
37          INSERTION IS AUTOMATIC
38          RETENTION IS FIXED;
39          DUPLICATES ARE NOT ALLOWED FOR LNO IN LA
40          SET SELECTION IS THRU A-LA OWNER
41              IDENTIFIED BY KEY A-SLT.
```

**Fig. 8.18 Het DBTG schema LEVERANCIERS-ARTIKELEN.**

```
MOVE    'L1' TO LNO IN L
FIND    ANY L USING LNO IN L
```

Andere manieren om records te lokaliseren zullen in hoofdstuk 10 besproken worden. Opgemerkt zij dat het in de DML opdracht "FIND ANY r USING x" *niet* verplicht is dat veld x als DUPLICATES NOT ALLOWED is gedeclareerd. Een mogelijke specificatie van x zou bijvoorbeeld kunnen luiden "DUPLICATES ARE SYSTEM DEFAULT", waarmee wordt aangegeven dat duplicaten aanwezig mogen zijn en dat die duplicaten in een door het DBMS bepaalde volgorde in de database worden opgeborgen. Deze volgorde is dan onbekend bij de gebruiker. Hierop komen we in paragraaf 10.11.1 terug.

Regels 5-8 definiëren de velden van het record L. Een veld wordt in een zogenaamde data subentry beschreven. Behalve het datatype CHARACTER is ook DECIMAL toegestaan (met een voor de hand liggende betekenis). In DBTG hoeft een veld *niet atomair* te zijn. Stel bijvoorbeeld dat het L record in plaats van een WNPL een ADRES zou kennen, met de volgende samengestelde structuur:

```
02  ADRES.
    03  POSTCODE       ;  TYPE IS CHARACTER 6.
    03  STRAAT         ;  TYPE IS CHARACTER 20.
    03  HUISNUMMER     ;  TYPE IS DECIMAL 3.
```

De nummers 02 en 03 zijn zogenaamde niveaunummers (level numbers) en worden gebruikt om de structuur van het veld ADRES vast te leggen (als alle velden atomair zijn, dan hebben niveaunummers geen zin). Hoe hoger het nummer (04, 05, ...) des te fijner de structuur. ADRES is blijkbaar opgebouwd uit drie 'deelvelden', namelijk een POSTCODE, een STRAAT en een HUISNUMMMER. Behalve samengestelde velden kan men ook repeterende groepen (repeating groups) beschrijven (zie ook hoofdstuk 4). In paragraaf 9.3 wordt hiervan een voorbeeld gegeven.

Regels 9-16 definiëren recordtype A op analoge wijze.

Regels 17-22 definiëren recordtype LA; merk op dat in dit geval de *combinatie* van twee velden als sleutel fungeert.

Regel 23 introduceert het settype L-LA, dat beschreven wordt in een zogenaamde 'set entry'. Voor L-LA omvat deze entry de regels 23 tot en met 32.

De regels 23 tot en met 25 beschrijven de zogenaamde set subentry van het settype L-LA. Regel 24 specificeert het owner-recordtype van L-LA, namelijk L.

Regel 25 legt vast dat de volgorde van LA records binnen elke L-LA set SORTED BY DEFINED KEYS is. Het "sorteercriterium", ofwel de te sorteren "sleutel" (Engels: "sort-key"), wordt door de KEY clausule (regel 29) gedefinieerd en is ASCENDING ANO IN LA. Met andere woorden,

voor elk L record zijn de LA records in de corresponderende L-LA set opklimmend (preciezer: niet-dalend) gesorteerd op artikelnummer. Het is de verantwoordelijkheid van het DBMS om deze volgorde gedurende het bestaan van de set te handhaven.

De regels 26 tot en met 32 beschrijven de member subentry van L-LA. Dit begint met het aanwijzen van het member-recordtype van L-LA, namelijk LA, in regel 26.

Regels 27-28 specificeren de MEMBERSHIP CLASS van LA in L-LA. In paragraaf 8.7 wordt dit begrip nader uitgelegd.

Aan regel 29 werd reeds aandacht besteed. In het algemeen kan een KEY clausule ASCENDING of DESCENDING zijn en de mag de "sort-key" een combinatie van een aantal velden in het record zijn. Een combinatie van opklimmende en afdalende sleutels kan worden gespecificeerd door een reeks ASCENDING/DESCENDING specificaties in de KEY clausule. De van-links-naar-rechts volgorde geeft het belang van de velden bij het sorteren aan (van belangrijk naar minder belangrijk).

Regel 30 (DUPLICATES ARE NOT ALLOWED) specificeert dat er in een L-LA set geen twee LA records mogen zitten met dezelfde ANO waarde.

Regel 31 betreft SET SELECTION waarvan de uitleg wordt uitgesteld tot paragraaf 8.8.

De regels 33-41 beschrijven het settype A-LA op analoge wijze. In regel 35 wordt de clausule ORDER IS FIRST gebruikt. Dit betekent dat een in te voegen record meteen na de owner wordt geplaatst (en dus voor alle reeds ingevoegde member-records komt te staan). ORDER IS LAST zorgt voor het tegenovergestelde. Andere ORDER varianten zijn PRIOR en NEXT. De set waar een record ingevoegd moet worden, kent een zogenaamd *current record*, dat is dat record van die set dat het laatst door de gebruiker (dat wil zeggen het programma) benaderd is (zie hoofdstuk 10). ORDER IS PRIOR (ORDER IS NEXT) bewerkstelligt dat het in te voegen record net vóór (ná) dit current record geplaatst wordt. Tenslotte geeft regel 39 de derde variant van de DUPLICATES clausule. Deze clausule geeft hier aan dat de member-records van een en dezelfde A-LA set een verschillende LNO waarde moeten hebben. Deze regel zal door het DBMS worden afgedwongen. (*Vraag*: er wordt in dit schema maar liefst driemaal geëist dat een combinatie LNO,ANO uniek moet zijn. Welke drie keer zijn dat?).

## 8.7 MEMBERSHIP CLASS

Elke member subentry in het schema moet een specificatie bevatten van de *membership class* voor het betreffende recordtype in het settype. De membership class wordt gespecificeerd door middel van de INSERTION/RETENTION clausule en mag daarom worden gezien als een combinatie van een insertion class en een retention class. De

retention class heeft te maken met het verwijderen van records uit een set, terwijl de insertion class juist het opnemen van records in een set betreft.

De insertion class is AUTOMATIC (automatisch) of MANUAL (manueel). De retention class is FIXED (gefixeerd), MANDATORY (verplicht) of OPTIONAL (optioneel). Een gegeven recordtype mag met betrekking tot een gegeven settype elke combinatie van een insertion class en een retention class hebben; bovendien mogen deze combinaties verschillend zijn voor verschillende settypen. Globaal gesproken beïvloedt de membership class van een member-recordtype de programma's die belast zijn met het onderhoud van het bijbehorende settype. Dat zijn dus de programma's die de sets van dat type onderhouden (records toevoegen, verwijderen of wijzigen). Met name worden de DML opdrachten CONNECT, DISCONNECT, RECONNECT, STORE en ERASE (en mogelijk MODIFY) door de membership class beïnvloed.

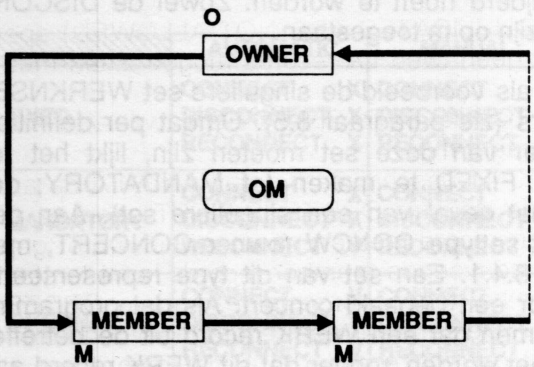



**Fig. 8.19 Typische set.**

Beschouw, om de gedachten te bepalen, een settype OM (owner O en member M). Figuur 8.19 geeft een voorbeeld van deze set.

- Retention class (FIXED, MANDATORY of OPTIONAL)

Als de retention class van M in OM FIXED is, dan houdt dat het volgende in. Als record m van type M member is van een set van type OM, dan blijft record m (zolang het in de database is opgenomen) in diezelfde set van OM zitten. Het record m kan niet door middel van een DISCONNECT operatie uit de set van type OM gehaald worden en m mag ook niet via een RECONNECT operatie naar een andere set van type OM verplaatst worden. De enige manier om de verbinding tussen m en de owner van de OM set te verbreken, is m door middel van een ERASE operatie uit de database te verwijderen. Merk op dat een belangrijk gevolg van de FIXED optie is, dat de verwijdering van een

record o (via een ERASE) de verwijdering van alle bij o horende records van type M inhoudt (*Vraag:* waarom?).

Als de retention class van M in OM MANDATORY is, dan betekent dat, dat zodra een record m van type M in een set OM is opgenomen, m member van *een* set van type OM moet blijven; dit hoeft niet steeds dezelfde set van type OM te zijn! Het record m kan nooit via een DISCONNECT operatie uit de set van type OM gehaald worden, omdat dat betekent dat m niet tegelijkertijd in een andere set van type OM wordt opgenomen. Dit in tegenstelling tot een RECONNECT, die ervoor zorgt dat m van de ene set van type OM in een andere set van type OM wordt verplaatst (in één ondeelbare actie). Een RECONNECT mag dus op m wel worden uitgevoerd.

Tenslotte, als de retention class van M in OM OPTIONAL is, dan kan m uit een set van type OM gehaald worden zonder dat m uit de database verwijderd hoeft te worden. Zowel de DISCONNECT als de RECONNECT zijn op m toegestaan.

Beschouw als voorbeeld de singuliere set WERKNSET, de set van alle werknemers (zie paragraaf 8.5). Omdat per definitie alle WERKN records member van deze set moeten zijn, lijkt het redelijk om de retention class FIXED te maken (of MANDATORY; deze twee zijn equivalent in het geval van een singuliere set). Aan de andere kant hebben we het settype CONCW (owner CONCERT, member WERK) van voorbeeld 8.4.1. Een set van dit type representeert een gepland programma voor een bepaald concert. Als dat programma wijzigt, dan kan het voorkomen dat een WERK record uit de betreffende CONCW set gehaald moet worden zonder dat dit WERK record aan een andere set van type CONCW toegevoegd moet worden (het kan immers voorkomen dat het werk helemaal niet gespeeld wordt). Het kan echter wenselijk zijn om het WERK record nog in de database te bewaren ten einde het later weer aan een andere CONCW set toe te voegen. Daarom lijkt het redelijk om de retention class OPTIONAL te maken.

- Insertion class (AUTOMATIC of MANUAL)

De insertion class zegt iets over de wijze waarop een record member van een set wordt. Als de insertion class van M in OM AUTOMATIC is, dan wordt een record m van type M op het moment van creatie (via een STORE operatie) reeds member van een set van type OM gemaakt; dit gebeurt door het DBMS zelf. Het is in het algemeen een taak van het programma dat m in de database laat opnemen om uit te maken in *welke* set van type OM record m moet worden opgenomen, zie paragraaf 8.8 (SET SELECTION). Als aan de andere kant de insertion class van M in OM MANUAL is, dan houdt het uitvoeren van een STORE niet een opname in een OM set in. Om m in een OM set te

laten opnemen moet het programma een expliciete CONNECT operatie geven.

Beschouw opnieuw de singuliere set WERKNSET van paragraaf 8.5. Omdat het niet mag voorkomen dat een WERKN record *geen* member van WERKNSET is, lijkt het redelijk om de insertion class AUTOMATIC te maken. Bij de set CONCW daarentegen (owner CONCERT, member WERK) kan het een eis zijn dat een WERK record in de database wordt opgenomen, zonder dat het meteen een member van een CONCW set wordt. In dit geval kan het gewenst zijn om *later*, zodra het werk opgenomen is in het programma van een concert, het WERK record in de juiste CONCW set op te nemen. Dus is het redelijk voor WERK een MANUAL insertion class in het settype CONCW te specificeren. Merk op dat dan de database naast informatie over concerten en componisten ook informatie over werken bevat die al gepland zijn of gepland zouden kunnen worden!

| | AUTOMATIC | MANUAL |
|---|---|---|
| FIXED | CONNECT X<br>DISCONNECT X<br>RECONNECT X | CONNECT V<br>DISCONNECT X<br>RECONNECT X |
| MANDATORY | CONNECT X<br>DISCONNECT X<br>RECONNECT V | CONNECT V<br>DISCONNECT X<br>RECONNECT V |
| OPTIONAL | CONNECT V<br>DISCONNECT V<br>RECONNECT V | CONNECT V<br>DISCONNECT V<br>RECONNECT V |

**Fig. 8.20 Het effect van membership class op CONNECT, DISCONNECT en RECONNECT.**

Figuur 8.20 geeft een gedeeltelijke samenvatting van het voorgaande. In deze figuur geeft een "V" aan dat een DML opdracht is toegestaan, terwijl een "X" de opdracht juist verbiedt. De ogenschijnlijke strijdigheid in deze figuur, namelijk dat CONNECT is toegestaan als de membership class OPTIONAL AUTOMATIC is, wordt verklaard door het feit dat na een automatische opname van een record in een set tijdens de uitvoering van de STORE operatie (insertion class is AUTOMATIC) een verwijdering uit die set kan volgen door de uitvoering van een DISCONNECT. Hierna kan een CONNECT volgen die het record weer in een set (mogelijk een andere) doet opnemen. Het punt is dat, terwijl de retention class (FIXED/MANDATORY/MANUAL) een tijdsonafhankelijke eigenschap is, dit niet geldt voor de insertion class (AUTOMATIC/MANUAL); deze heeft alleen betekenis op het moment

van de opname van het record in de database (via de STORE operatie) en is niet relevant op een later tijdstip.

## 8.8 SET SELECTION

Er zijn bepaalde situaties waarin het DBMS in staat moet zijn om een set automatisch te selecteren. Een van die situaties is reeds in de vorige paragraaf genoemd en doet zich voor zodra een nieuw record in de database wordt opgenomen en het record van een type is dat AUTOMATIC member is van een of meer settypen. Voor elk van deze settypen moet het DBMS een set bepalen waar het betreffende record aan toegevoegd moet worden. We zullen in hoofdstuk 10 nog andere voorbeelden van SET SELECTION tegenkomen.

Om het DBMS deze automatische selectie goed te kunnen laten uitvoeren, moet de DBA een SET SELECTION clausule specificeren in de (MEMBER subentry van de) setentry van het schema. Deze paragraaf bestaat uit een enigsins vereenvoudigde uitleg van deze clausule. Laten we ons even beperken tot het settype L-LA (zie paragraaf 8.6). Beschouw de situatie waarbij het nodig is om het nieuwe LA record "L5/A6/700" in de database op te nemen (via een STORE natuurlijk). Opgemerkt zij dat LA in Fig. 8.18 als een AUTOMATIC member van L-LA is gedefinieerd. Dit geldt natuurlijk ook voor A-LA, maar dat vergeten we even.

In het *eerste* geval ziet de SET SELECTION clausule voor het settype L-LA (met member LA) er als volgt uit:

```
SET SELECTION IS THRU L-LA OWNER
    IDENTIFIED BY APPLICATION.
```

Dit betekent dat het toepassingsprogramma verantwoordelijk is voor het selecteren van de correcte L-LA set. Dit selecteren gebeurt voordat het nieuwe LA record in de database wordt opgenomen. Het DBMS neemt aan dat de L-LA set waarvan het laatst een owner- of member-record geselecteerd is, ook de set is waar het LA record ingevoegd moet worden. Het programma zal dus normaal gesproken de set, waarin het LA record moet worden opgenomen, selecteren voordat een STORE wordt gegeven. Dit selecteren gebeurt door een andere operatie, namelijk de FIND opdracht. Een van de mogelijke manieren om het LA record "L5/A6/700" in de database op te nemen, zou dan kunnen zijn:

```
bouw LA record 'L5/A6/700' op in UWA
MOVE    'L5' TO LNO IN L
FIND    ANY L USING LNO IN L
STORE   LA
```

In het *tweede* geval ziet de SET SELECTION clausule er als volgt uit (zie ook het schema van Fig. 8.18):

```
SET SELECTION IS THRU L-LA OWNER
      IDENTIFIED BY KEY L-SLT
```

Dit betekent dat het DBMS de L-LA set bepaalt door met gebruikmaking van de waarde van L-SLT het owner-record van de set te lokaliseren. L-SLT moet wel als DUPLICATES ARE NOT ALLOWED zijn gespecificeerd (*Waarom?*). Het DBMS neemt de L-SLT waarde uit de UWA. Als de owner bepaald is, dan is ook de gewenste set bepaald.

In ons voorbeeld moet de programmeur dus het veld LNO IN L in de UWA van de waarde "L5" voorzien voordat de STORE opdracht wordt gegeven:

```
bouw LA record 'L5/A6/700' op in UWA
MOVE    'L5' TO LNO IN L
STORE   LA
```

Een variant op de hiervoor genoemde clausule is de volgende:

```
SET SELECTION IS THRU L-LA OWNER
      IDENTIFIED BY KEY L-SLT
      LNO IN L EQUAL TO VELD-1
```

Nu moet het DBMS niet de waarde van het UWA-veld LNO IN L (dat de sleutel vormt) nemen, maar juist de waarde van VELD-1.

De *derde* en laatste vorm van de SET SELECTION is een zeer eenvoudige en mag alleen worden gebruikt bij een singuliere set. Een dergelijk settype heeft slechts één exemplaar en vanzelfsprekend moet bij SET SELECTION altijd dit ene exemplaar door het DBMS worden gekozen. Dit wordt voor de singuliere set AFDSET aangegeven door:

```
SET SELECTION IS THRU AFDSET OWNER
      IDENTIFIED BY SYSTEM
```

De afhandeling van de SET SELECTION heeft geen invloed op de zogenaamde currency indicators. Deze indicators worden in hoofdstuk 10 besproken.

## 8.9 VERSCHILLEN TUSSEN SUBSCHEMA EN SCHEMA

In DBTG wordt een externe visie gedefinieerd door middel van een *subschema*. Met de schema DDL kan men de gegevens beschrijven die in het object systeem een rol spelen. Als zodanig moet de schema DDL dan ook onafhankelijk zijn van de programmeertalen die op extern niveau worden gebruikt. Dit geldt niet voor de subschema DDL en de DML. In de rest van dit hoofdstuk onderzoeken we de COBOL subschema DDL, terwijl in hoofdstuk 10 de COBOL DML aan de orde komt. Globaal gesproken is een subschema een deelverzameling van het corresponderende schema.

Bij een gegeven schema mogen vele subschema's gedefinieerd worden. Meerdere programma's mogen een gegeven subschema gebruiken (zie Fig. 8.1); verschillende subschema's mogen overlappen. Merk op, dat het de "aanroep" van een subschema is waardoor een programma de definitie van de "user work area" (UWA) aangeboden krijgt. Een voorbeeld van zo'n aanroep is

```
DB   LEVERANCIERS WITHIN LEVERANCIERS-ARTIKELEN.
```

Deze opdracht (die in de DATA DIVISION van het COBOL programma voorkomt) roept het subschema met de naam LEVERANCIERS aan. Dit subschema is van het schema LEVERANCIERS-ARTIKELEN afgeleid. We zullen in paragraaf 8.10 een mogelijke definitie van LEVERANCIERS laten zien.

Laten we eerst de bewering dat het subschema een deelverzameling van het schema is, nader preciseren. In het algemeen mag elk van de volgende schema-delen in een subschema worden weggelaten:

- de declaratie van een of meer settypen,
- de declaratie van een of meer recordtypen, en
- de declaratie van een of meer veldtypen.

Natuurlijk moet het subschema consistent zijn. Zo moet de declaratie van een recordtype niet ontbreken als er een settype in het subschema voorkomt, dat dat recordtype als owner of member heeft.

Uit het voorgaande volgt dat gebruikers afgeschermd zijn van zekere vormen van "groei" in het schema. Zo kunnen bijvoorbeeld nieuwe velden worden toegevoegd aan een bestaand recordtype. Evenzo kunnen nieuwe recordtypen en (onder zekere voorwaarden) nieuwe settypen worden toegevoegd.

Hieruit volgt dat het subschema automatisch voorziet in een zekere vorm van beveiliging van de gegevens en wel in die zin, dat een programma onmogelijk toegang kan krijgen tot gegevens die niet in het subschema zijn gedefinieerd (uitgezonderd in het geval van ERASE, zie hoofdstuk 10).

Andere belangrijke verschillen die tussen subschema en schema mogen bestaan zijn de volgende.

- Eigen namen (zogenaamde aliassen) mogen voor settypen, recordtypen en veldtypen worden gedefinieerd.
- Veldtypen mogen een verschillend data type hebben.
- De relatieve volgorde van veldtypen binnen het omvattende recordtype mag verschillend zijn.
- Settypen mogen verschillende SET SELECTION clausules hebben.

Het laatste verschil vereist enige nadere uitleg. Het idee is, dat de SET SELECTION clausule in het schema overschreven kan worden door een SET SELECTION clausule in het subschema. In het schema zou BY APPLICATION gespecificeerd kunnen zijn, inhoudend dat de set selectie procedureel in het programma geregeld moet worden, terwijl in het subschema BY VALUE gespecificeerd zou kunnen zijn. Dit laatste betekent dat programma's die dat subschema aanroepen, de set selectie wat minder procedureel regelen.

## 8.10 EEN VOORBEELD SUBSCHEMA

Figuur 8.21 geeft een eenvoudig voorbeeld van een COBOL subschema dat gebaseerd is op het schema van Fig. 8.18. Eigenlijk bevat dit subschema alleen de gegevens over leveranciers met bijbehorende leverbare artikelen en worden de gegevens over artikelen uitgesloten.

In het algemeen bestaat een COBOL subschema uit een Title Division, die de namen van het subschema en het onderliggende schema geeft, een Mapping Division (waarin de aliassen zijn gedefinieerd) en een Structure Division. De Structure Division bestaat op zijn beurt uit drie secties: een Realm Section, een Record Section en een Set Section. De Realm Section wordt hierna uitgelegd. De Record Section geeft alle van belang zijnde schema recordtypen (met de benodigde veldtypen). De Set Section geeft alle van belang zijnde settypen uit het schema.

De Realm Section specificeert de van belang zijnde *realms*. Een realm is een logische deelverzameling van de database zoals die in het subschema is vastgelegd. Het bestaat uit alle records van een of meer gespecificeerde typen (heel vaak zijn dit alle typen die in de Record Section worden genoemd; dit is ook het geval in Fig. 8.21). Een gegeven subschema kan meerdere realms bevatten, waarbij verschillende realms kunnen overlappen (dat wil zeggen dat ze recordtypen gemeenschappelijk hebben).

```
TITLE DIVISION.

  SS  LEVERANCIERS WITHIN LEVERANCIERS-ARTIKELEN.

MAPPING DIVISION.

  ALIAS SECTION.
  AD  SET LEVERT IS L-LA.

STRUCTURE DIVISION.

  REALM SECTION.
  RD  LEVERT-REALM CONTAINS L, LA RECORDS.

  RECORD SECTION.
  01  L.
       02  LNO     ; PICTURE IS X(5).
       02  WNPL    ; PICTURE IS X(24).
  01  LA.
       02  ANO     ; PICTURE IS X(6).
       02  LNO     ; PICTURE IS X(5).
       02  HOEV    ; PICTURE IS S9(5).

  SET SECTION.
  SD  LEVERT.
```

**Fig. 8.21 Een voorbeeld subschema (COBOL).**

Elk realm specificeert een "verwerkingsgebied". Voordat een programma toegang tot een deel van de in het subschema gedefinieerde database kan krijgen, moet het eerst een READY operatie voor de betreffende realm uitvoeren. Door deze READY operatie wordt het DBMS geïnformeerd over het feit dat toegang tot dat deel van de database gewenst is. De READY opdracht geeft ook aan welke vorm van toegang gewenst is: blijft de toegang beperkt tot leesoperaties of wordt de database ook bijgewerkt. Meer details volgen in hoofdstuk 10.

## OPGAVEN

**8.1** Bedenk realistische voorbeelden voor ieder van de INSERTION/RETENTION combinaties.

**8.2** In dit hoofdstuk zijn drie SET SELECTION clausules besproken. Geef in deze drie gevallen aan wat een programma moet doen om de selectie van de set te specificeren.

**8.3** Gegeven is een eenvoudige database bevattende gegevens over boeken, leners en auteurs. Boeken zijn geschreven door één auteur en mogen zijn uitgeleend aan één lener. Het settype GESCHRIFT kent als owner het recordtype AUTEUR en als member het recordtype BOEK; de membership class is AUTOMATIC/FIXED. Het settype LENING heeft als membership class OPTIONAL/MANUAL (het zogenaamde sometime membership); LENER is het owner en BOEK het member-recordtype. Geef het Bachman data structuurdiagram voor deze database (geef het sometime membership weer door een stippellijn). Beantwoord vervolgens de volgende vragen:

a. Kan een auteur geen boeken hebben geschreven?
b. Kan een boek geen auteur hebben?
c. Kan een lener geen boeken hebben geleend?
d Kan een boek op dit moment geen lener hebben?
e. Kan het voorkomen dat een boek nooit wordt uitgeleend?
f. Kan een lener alle boeken van een bepaalde auteur geleend hebben?

**8.4** Definieer een schema voor de management database van paragraaf 8.3.4.

**8.5** Definieer een schema voor de leveranciers-artikelen-projecten database (diverse opgaven van hoofdstuk 10 zijn op dit schema gebaseerd).

**8.6** Definieer een schema voor de artikelen database (Fig. 8.14).

**8.7** Veronderstel dat een nieuw settype OM (owner O en member M) aan het schema wordt toegevoegd. In hoeverre kunnen bestaande programma's door deze toevoeging ongewijzigd blijven? U moet hierbij de volgende gevallen beschouwen.

a) O en M zijn beide nieuwe recordtypen (ook toegevoegd)
b) O en M zijn beide oude (bestaande) recordtypen
c) O is oud en M is nieuw
d) O is nieuw en M is oud.

(Merk op dat de membership class hier belangrijk is, evenals de SET SELECTION clausule. Het is wellicht verstandig de beantwoording van deze vraag uit te stellen totdat hoofdstuk 10 is behandeld en de paragrafen 8.7 en 8.8 nog eens zijn doorgenomen.)

# 9
# Ontwerp van DBTG Databases

## 9.1 INLEIDING

In dit hoofdstuk wordt aangegeven hoe de begrippen entiteit, associatie en kenmerk uit het object systeem in het DBTG data model kunnen worden gerepresenteerd. De besproken ontwerpmethode heeft veel verwantschap met de relationele ontwerpmethode die in hoofdstuk 5 werd besproken. Achtereenvolgens komen aan de orde: het representeren van entiteittypen, n:1 associatietypen, n:m associatietypen en niet-atomaire kenmerken. Bovendien wordt weer aandacht besteed aan faciliteiten om de integriteit van een database te handhaven. Dit hoofdstuk heeft, net als de hoofdstukken 5 en 6, niet zozeer te maken met de eigenschappen van een data model, maar meer met het gebruik ervan.

## 9.2 ENTITEITEN EN ASSOCIATIES

We gaan weer uit van het probleem van hoofdstuk 5, namelijk het ontwerpen van een bibliotheeksysteem waarin gegevens over boeken moeten worden bijgehouden. Van deze boeken zijn titel, auteurnaam (of auteurnamen) en inhoudsopgave van belang. Als een boek meerdere auteurs heeft, dan willen we weer de volgorde van de auteurs weten. Een inhoudsopgave bevat weer informatie over de hoofdstukken van het boek. Bovendien hoort er bij ieder boek één uitgever en moet van een uitgever naam, adres en type worden vastgelegd. Verder worden van de auteurs enige nadere gegevens bijgehouden.

**9.2.1 Entiteiten.** In dit voorbeeld komt een aantal entiteiten voor. We hebben reeds gezien hoe entiteittypen in het DBTG data model worden weergegeven.

- *In het DBTG data model worden entiteiten van hetzelfde type door records van hetzelfde type gerepresenteerd.*

We zien in het bibliotheeksysteem de volgende entiteittypen: Boeken, Uitgevers en Auteurs. Deze worden dus als recordtypen weergegeven.

```
RECORD NAME IS BOEKEN;
    KEY BOEK-SLT IS BOEKNO IN BOEKEN
        DUPLICATES ARE NOT ALLOWED.
    BOEKNO        ; TYPE IS CHARACTER 8.
    PLAATSNUMMER  ; TYPE IS CHARACTER 10.
    TITEL         ; TYPE IS CHARACTER 40.
    INHOUDSOPGAVE ; (* zie paragraaf 9.3 *).

RECORD NAME IS UITGEVERS;
    KEY UITG-SLT IS UITG-NAAM IN UITGEVERS
        DUPLICATES ARE NOT ALLOWED.
    UITG-NAAM     ; TYPE IS CHARACTER 25.
    UITG-ADRES    ; TYPE IS CHARACTER 30.
    TYPE          ; TYPE IS CHARACTER 2.

RECORD NAME IS AUTEURS;
    KEY AUT-SLT IS AUT-NAAM IN AUTEURS
        DUPLICATES ARE NOT ALLOWED.
    AUT-NAAM      ; TYPE IS CHARACTER 25.
    AUT-ADRES     ; TYPE IS CHARACTER 30.
```

**9.2.2 Associaties.** In hoofdstuk 8 zagen we aan de hand van enige voorbeelden hoe n:1 associatietypen worden geïmplementeerd.

- *In het DBTG data model wordt een veel-op-een associatietype door een settype gerepresenteerd.*

In ons geval betekent dit, dat een settype UITGAVE voor het associatietype tussen de entiteittypen Uitgevers en Boeken ingevoerd moet worden:

```
SET NAME IS UITGAVE;
    OWNER IS UITGEVER;
    ...
    MEMBER IS BOEKEN;
    ...
```

Veel-op-veel associatietypen kwamen in hoofdstuk 8 een aantal keren voor (zie Fig. 8.12, 8.14 en het voorbeeld van de leveranciers-artikelen database). Het volgende bleek:

- *Veel-op-veel associatietypen worden in het DBTG data model door een recordtype plus twee settypen gerepresenteerd.*

In ons voorbeeld hebben we een veel-op-veel associatietype tussen Auteurs en Boeken. Een auteur kan aan vele boeken meewerken en een boek kan door een aantal auteurs zijn geschreven. Dit wordt als volgt weergegeven:

```
RECORD NAME IS AUTEUR-BOEK;
     KEY AB-SLT IS AUT-NAAM IN AUTEUR-BOEK,
                   BOEKNO IN AUTEUR-BOEK
         DUPLICATES ARE NOT ALLOWED.
     AUT-NAAM       ; TYPE IS CHARACTER 25.
     BOEKNO         ; TYPE IS CHARACTER 8.
     VOLGNO         ; TYPE IS CHARACTER 1.

SET NAME IS OEUVRE;
     OWNER IS AUTEUR;
     ...
     MEMBER IS AUTEUR-BOEK;
     ...

SET NAME IS SCHRIJVERS;
     OWNER IS BOEKEN;
     ...
     MEMBER IS AUTEUR-BOEK;
     ...
```

**9.2.3 Recursieve settypen.** We gaan er vanuit dat in het DBTG database-ontwerp recursieve settypen (nog) niet mogen worden gebruikt. Door een extra recordtype in te voeren, wordt vermeden dat owner en member van hetzelfde recordtype zijn. Het voorbeeld van de bibliotheek bevat geen recursieve settypen; in paragraaf 8.3.4 is echter wel een voorbeeld van recursieve settypen behandeld.

## 9.3 NIET-ATOMAIRE KENMERKEN

Aangezien in DBTG zogenaamde repeterende groepen zijn toegestaan, kan een niet-atomair kenmerk direct op zo'n niet-atomair veld afgebeeld worden.

- *In het DBTG data model worden niet-atomaire kenmerken van entiteiten door samengestelde velden van records gerepresenteerd.*

Hieronder wordt een voorbeeld gegeven van een niet-atomair kenmerk, namelijk de inhoudsopgave van het entiteittype Boeken. We nemen aan

dat een inhoudsopgave maximaal 20 hoofdstukken mag bevatten. Per hoofdstuk wordt het een en ander bijgehouden; de inhoudsopgave is dus een niet-atomair kenmerk van boeken.

Een niet-atomair kenmerk kan op een samengesteld veld worden afgebeeld. De opbouw van zo'n samengesteld veld wordt weergegeven door niveaunummers, waarover in paragraaf 8.6 reeds het een en ander werd uitgelegd. Voor het kenmerk inhoudsopgave wordt hieronder de opbouw beschreven. Zo is het veld HFDST-NO (niveaunummer 03) een deel van (het samengestelde veld) INHOUDSOPGAVE (niveaunummer 02). Achter INHOUDSOPGAVE staat "OCCURS 20 TIMES", waarmee wordt aangegeven dat een INHOUDSOPGAVE uit 20 deelstructuren bestaat (iedere deelstructuur beschrijft een hoofdstuk). (De datatypen zijn hieronder voor de duidelijkheid wel gegeven.) De rest van het voorbeeld spreekt voor zich.

```
RECORD NAME IS BOEKEN;
    KEY BOEK-SLT IS BOEKNO IN BOEKEN
      DUPLICATES ARE NOT ALLOWED.
    02  BOEKNO          ; TYPE IS CHARACTER 8.
    02  PLAATSNUMMER    ; TYPE IS CHARACTER 10.
    02  TITEL           ; TYPE IS CHARACTER 40.
    02  INHOUDSOPGAVE OCCURS 20 TIMES.
      03  HFDFST-NO        ; TYPE IS CHARACTER 2.
      03  HFDST-TITEL      ; TYPE IS CHARACTER 40.
      03  HFDST-PAGNO      ; TYPE IS CHARACTER 4.
```

Opmerking.
In het relationele model worden niet-atomair kenmerken van hetzelfde type gerepresenteerd door een relatie met een vreemde sleutel. Deze aanpak kan in het DBTG model nagebootst worden door een extra recordtype met een settype te introduceren. Geef zelf aan hoe een en ander er dan voor het niet-atomaire kenmerk INHOUDSOPGAVE uit gaat zien.

## 9.4 INTEGRITEIT EN BIJWERKOPERATIES

Het DBTG data model kent settypen waarmee n:1 associatietypen kunnen worden geïmplementeerd. Zoals in het vorige hoofdstuk reeds is aangegeven, kent het DBTG model restricties ten aanzien van het opereren op sets. De belangrijkste clausule in deze is de MEMBERSHIP CLASS, die aangeeft wanneer een record in een set wordt opgenomen en wanneer een set wordt verlaten. Net als in paragraaf 5.4 gaan we eerst kijken wat de gevolgen met betrekking tot integriteit kunnen zijn bij het invoegen en verwijderen van een owner record. Daarna beschouwen we het invoegen en verwijderen van een member uit een set.

**9.4.1 Invoegen van record.** Als een record een sleutel heeft (bijvoorbeeld "KEY L-SLT IS LNO IN L; DUPLICATES ARE NOT ALLOWED"), dan zal het DBMS het uniek zijn van de sleutelwaarde van een in te voegen record controleren en, indien nodig, de invoegactie weigeren (invoegen van een record gaat via een STORE, zie paragraaf 10.5).

**9.4.2 Verwijderen van record.** Wat gebeurt er als de owner van een set verwijderd wordt (via een ERASE opdracht, zie paragraaf 10.6).

Het blijkt dat de ingevoerde begrippen met betrekking tot MEMBERSHIP CLASS, en dan nog meer specifiek de 'retention class', een belangrijke rol spelen bij de bepaling van de semantiek van de ERASE opdracht. In het DBTG data model wordt de semantiek van de ERASE bepaald door de retention class van de set(s) (FIXED, MANDATORY of OPTIONAL) waar het te verwijderen record owner van is:

- Voor FIXED members geldt dat de ERASE "cascadeert", dat wil zeggen: de ERASE wordt ook toegepast op alle records die member van een FIXED settype zijn.
- Voor MANDATORY members: als er records zijn die member zijn van MANDATORY settypen dan gaat de ERASE in zijn totaliteit niet door.
- Voor OPTIONAL members: als er records zijn die member zijn van OPTIONAL settypen, dan worden deze records uit de betreffende sets gehaald, maar blijven bestaan.

Let wel, als een te verwijderen record FIXED members heeft, dan wordt de ERASE ook op die members toegepast (recursiviteit). Als een van die FIXED members owner is van een set met MANDATORY members, dan gaat de *gehele* ERASE niet door! In paragraaf 10.6 volgt een nadere uitleg.

**9.4.3 Opname in een set.** Als de insertion class van de MEMBERSHIP CLASS van een settype AUTOMATIC is, dan wordt een member bij opname in de database ook member van een set van dat type. Dat houdt in dat de STORE een opname in een set bewerkstelligd. Een en ander is in paragraaf 8.7 uitgelegd.

Aan de andere kant houdt MANUAL in dat een programmeur expliciet via een CONNECT (zie 10.8) een record in een set moet doen opnemen.

**9.4.4 Verlaten van een set.** De retention class bepaalt de wijze waarop een set kan worden verlaten. FIXED geeft aan dat een eenmaal ingenomen set niet meer mag worden verlaten voordat het member uit de database wordt verwijderd. OPTIONAL houdt in dat het record vrijelijk

een set mag verlaten (via een DISCONNECT, zie 10.9) en weer member van een andere set van dat type mag worden (via een CONNECT). MANDATORY betekent dat een record na member van een set geworden te zijn, altijd member van de een of andere set van dat type moet zijn (van set wisselen gaat via een RECONNECT, zie 10.10)

## 9.5 SLOTOPMERKINGEN

De aandachtige lezer heeft mogelijk al opgemerkt dat er grote overeenkomst bestaat tussen de restricties ten aanzien van het manipuleren op vreemde sleutels (zie de paragrafen 5.4.3 en 5.4.4) en de MEMBERSHIP CLASS van het DBTG data model. Meer specifiek hebben aan de ene kant de retention class en de MODIFY restricties met elkaar te maken, en aan de andere kant de insertion class en de INSERT restricties. Laten we eerst de retention class en de MODIFY restricties beschouwen.

- "MODIFY ALLOWED" komt overeen met retention class OPTIONAL, aangezien beide clausules inhouden dat een bestaande associatie tussen twee records (tupels) zonder restricties verbroken mag worden.
- "MODIFY ALLOWED EXCEPT TO NULL" en retention class MANDATORY geven ook hetzelfde aan, namelijk dat als een record (tupel) eenmaal een bepaalde associatie is aangegaan, het altijd moet participeren in een associatie van dit type.
- "MODIFY ALLOWED ONLY FROM NULL" en retention class FIXED betekenen hetzelfde: als een record (tupel) eenmaal in een bepaalde associatie participeert, dan blijft dat zo totdat het record (tupel) uit de database wordt verwijderd.

Ten aanzien van de insertion class en de INSERT restricties valt de volgende analogie op te merken.

- "ON INSERT NULL ALLOWED" en insertion class is MANUAL geven hetzelfde weer: een associatie tussen de twee entiteiten hoeft niet meteen bij het opnemen in de database te worden vastgelegd. Dit kan later gebeuren.
- Evenzo hebben "ON INSERT NULL NOT ALLOWED" en insertion class AUTOMATIC dezelfde semantiek: zodra de representaties van de twee entiteiten in de database zijn opgenomen, moet de associatie worden vastgelegd.

De DELETE en ERASE opdracht kennen een belangrijk verschil ten aanzien van de integriteit. In het relationele data model geeft de database-ontwerper de semantiek van de DELETE *voor iedere vreemde sleutel* aan met behulp van de termen CASCADES, RESTRICTED en NULLIFIES (zie paragraaf 5.4.2). Dit kan onafhankelijk van de INSERT en MODIFY restricties van de betrokken vreemde sleutel gebeuren! In

het DBTG model kent de ERASE een vaste semantiek afhankelijk van de retention class van de sets waar het te verwijderen record owner is. De voorgestelde uitbreiding van het relationele model is op dit punt dus flexibeler dan het DBTG model.

Tenslotte nog een opmerking over implementaties. Terwijl de DELETE, INSERT en MODIFY opties in het relationele systeem slechts zelden in relationele database systemen worden ondersteund, wordt de MEMBERSHIP CLASS niet alleen al jaren in de DBTG specificaties genoemd, maar ook in nagenoeg alle DBTG systemen aangeboden. Op dit punt zijn de DBTG systemen hun relationele tegenhangers dus vooruit!

## OPGAVEN

**9.1** Geef de membership classes voor de settypen OEUVRE en SCHRIJVERS van paragraaf 9.2.

**9.2** Beschouw opnieuw de opgaven 5.2, 5.3 en 5.4. Nu wordt echter gevraagd om het DBTG schema voor deze database te ontwerpen in plaats van een relationeel schema.

# 10
# Manipulatie van een DBTG Database

## 10.1 INLEIDING

Net als bij de subschema DDL zullen we de uitleg van de DBTG data manipulatietaal (DML) baseren op COBOL. Een volledige syntax-beschrijving van deze DML vindt u in [KLUW87], terwijl het gedeelte dat in dit hoofdstuk behandeld wordt in Appendix C wordt samengevat.

De inhoud van dit hoofdstuk ziet er als volgt uit. Paragraaf 10.2 behandelt het begrip 'currency', een begrip dat wezenlijk is voor het begrijpen van DML opdrachten. Paragraaf 10.3 bespreekt de behandeling van fouten en uitzonderlijke situaties die tijdens de uitvoering van een programma kunnen optreden. De paragrafen 10.4-10.11 behandelen alle belangrijke componenten van de DML. De FIND opdracht wordt het laatst behandeld (in paragraaf 10.11). Deze opdracht is de moeilijkste, maar tevens de belangrijkste DML opdracht. De paragrafen 10.12 en 10.13 geven dan nog een korte beschrijving van enige overblijvende opdrachten. In paragraaf 10.14 wordt tenslotte ingegaan op enige database-condities. Dat zijn condities die gerelateerd zijn aan de toestand van de database en die bijvoorbeeld gebruikt kunnen worden in een IF opdracht van COBOL.

Alle voorbeelden zullen worden gebaseerd op Fig. 8.15 (die voor het gemak van de lezer in Fig. 10.1 nog eens wordt getoond) en het schema van Fig. 8.18 (van de leveranciers-artikelen database). We nemen aan dat het subschema gelijk is aan het schema; het subschema wordt aangeroepen zoals in paragraaf 8.9 is aangegeven.

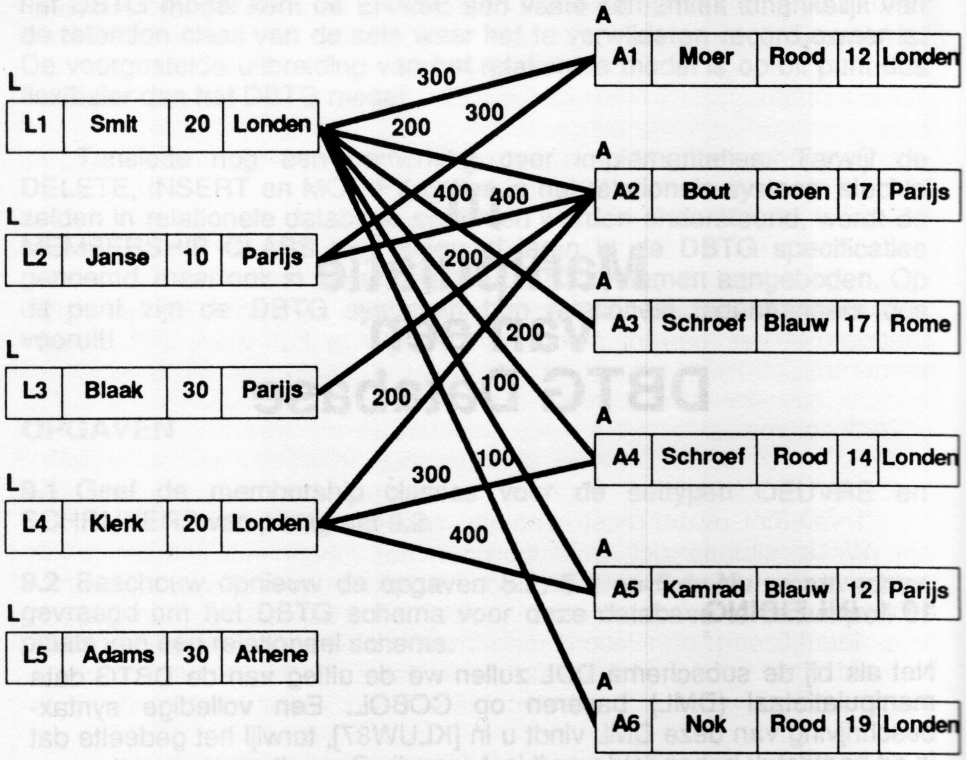



**Fig. 10.1 Voorbeeld gegevens (leveranciers en artikelen).**

## 10.2 CURRENCY

Voordat we de DML opdrachten gedetailleerder bekijken, is het nodig om het fundamentele begrip 'currency' te bespreken. Het idee is dat voor ieder programma dat met de database werkt, het DBMS een tabel met 'currency indicators' bijhoudt. Een currency indicator is een variabele die waarden van het type *database-sleutel* (database key) kan aannemen. Database-sleutels zijn door het DBMS gegenereerde waarden die records in de database identificeren (een database-sleutel kan men zich voorstellen als het adres van het opgeslagen record). Globaal gesproken identificeren de currency indicators tijdens de uitvoering van een programma de records die het laatst benaderd zijn (zoals we in deze paragraaf nog zullen zien is de situatie iets complexer). Voor elk van de

hieronder genoemde database-objecten is een currency indicator aanwezig.

- Elk realm

Voor ieder realm M wordt het record dat het laatst benaderd is het "current record van realm M" genoemd.

- Elk recordtype

Voor ieder recordtype R wordt het record van type R dat het laatst benaderd is het "current record van type R" of het "current R record" genoemd.

- Elk settype

Voor een settype S mag het laatst benaderde record zowel een record van het owner recordtype als een van het member recordtype zijn. Welke situatie zich ook voordoet, het record wordt het "current record van set S" genoemd. Merk op, dat het "current record van set S" een *record* is, maar dat daarnaast dat record ook een *set* van type S uniek identificeert: een record mag immers slechts in één set van een bepaald type voorkomen. (Waarom is dat zo?) De set die door het current record van set S wordt geïdentificeerd, noemt men "current S set".

- De run-unit

Tijdens de uitvoering van een programma (zo'n uitvoering wordt in DBTG termen *run-unit* genoemd) worden vele records benaderd. Het record dat het meest recent benaderd is (in welk realm dit record ook opgeslagen is en van welke sets het ook member is), wordt het "current record van de run-unit" genoemd (hetgeen normaal wordt afgekort tot "current van run-unit"). De bijbehorende currency indicator is de belangrijkste.

Beschouw de volgende rij opdrachten.

```
        MOVE    'L4' TO LNO IN L
1       FIND    ANY L USING LNO IN L
2       FIND    FIRST LA WITHIN L-LA
                RETAINING L-LA CURRENCY
3       FIND    OWNER WITHIN A-LA
```

Het effect van deze opdrachten is de volgende (zie Fig. 10.1). De MOVE initialiseert het UWA veld LNO IN L. De eerste FIND lokaliseert het corresponderende L record (hetgeen betekent dat de database-sleutel van het record van de leverancier L4 wordt bepaald; de uitvoering

van een FIND houdt *geen* data transport van database naar UWA in; daar zorgt een GET voor!). Door deze FIND worden de noodzakelijke currency indicators bijgewerkt, namelijk van run-unit, L recordtype, L-LA settype en van realm L-LA-A. Bovendien wordt de current L-LA set bepaald (zie de tabel hierna). De volgende FIND lokaliseert het eerste LA record in de L-LA set, waarvan het L4 record owner is. Dit is het LA record 'L4/A2/200'. De RETAINING clausule zorgt ervoor dat de currency indicator van het L-LA settype *niet* wordt bijgewerkt. In paragraaf 10.11.7 zullen we het nut van een dergelijke clausule zien. De laatste FIND lokaliseert tenslotte de owner van de A-LA set van dit LA record, namelijk het A record van artikel A2. Aan het eind van deze rij opdrachten is dit A record dan ook de current van run-unit.

In de tabel hierna worden alle currency indicators van de leveranciers-artikelen database gegeven. Bovendien worden na iedere FIND de records gegeven waar (via de database-sleutel) naar verwezen wordt (aangenomen wordt dat het realm L-LA-A de gehele database bevat). Er worden ook twee sets aangegeven, die na iedere FIND opdracht current van respectievelijk L-LA en A-LA zijn. Dit gebeurt door de owner van de set te noemen!

| Na uitvoering van opdracht | (1) | (2) | (3) |
|---|---|---|---|
| Current van run-unit | 'L4' | 'L4/A2 | 'A2' |
| Current L record | 'L4' | 'L4' | 'L4' |
| Current A record | --- | --- | 'A2' |
| Current LA record | --- | 'L4/A2' | 'L4/A2' |
| Current record van settype L-LA | 'L4' | 'L4' | 'L4' |
| Current record van settype A-LA | --- | 'L4/A2' | 'L4/A2' |
| Current L-LA set; *owner*: | 'L4' | 'L4' | 'L4' |
| Current A-LA set; *owner*: | --- | 'A2' | 'A2' |
| Current record van realm L-LA-A | 'L4' | 'L4/A2' | 'A2' |

De volgende samenvatting van de belangrijkste DML opdrachten geeft aan waarom currency (en in het bijzonder "current van run-unit") zo'n belangrijk begrip is.

- FIND
  lokaliseert een bestaand record en maakt het tot current van run-unit (waarbij ook de noodzakelijke andere currency indicators worden bijgewerkt).
- GET
  leest de current van run-unit, dat wil zeggen, brengt de inhoud van dat database record in de UWA.
- ERASE
  verwijdert de current van run-unit uit de database.
- STORE
  creëert een nieuw record, zet het in de database en maakt het tot current van run-unit (waarbij ook de noodzakelijke andere currency indicators worden bijgewerkt).
- MODIFY
  werkt de current van run-unit in de database bij.
- CONNECT
  neemt de current van run-unit op in een set.
- DISCONNECT
  haalt de current van run-unit uit een set.
- RECONNECT
  haalt de current van run-unit uit een set en neemt hetzelfde record weer op als een member van een (meestal andere) set van hetzelfde type.

Het belang van de FIND volgt duidelijk uit deze samenvatting. De FIND is vereist voor elk van de andere opdrachten, behalve de STORE. We zullen echter de behandeling van de FIND, die vele vormen (in het Engels is vorm "format") kent, uitstellen tot aan het eind van dit hoofdstuk. Als de uitleg van de opdrachten de uitvoering van een FIND opdracht vereisen, zullen we ons beperken tot reeds gedemonstreerde, eenvoudige vormen van de FIND opdracht.

Deze paragraaf over currency sluiten we af met de behandeling van een opdracht waarmee een currency indicator kan worden toegekend aan een variabele uit het COBOL programma.

```
MOVE CURRENCY STATUS FOR RUN-UNIT TO identifier
```

Hierbij is "identifier" de naam van een COBOL variabele, die van het datatype database-sleutel moet zijn. Zoals gezegd geeft een currency

indicator een 'positie' in de database aan. Als we later naar deze positie willen terugkeren, dan is het gemakkelijk om deze zolang te bewaren in een COBOL variabele. We zullen in 10.11.6 zien dat er een FIND variant is die het 'omgekeerde' bewerkstelligt. Dat wil zeggen een variant die, gegeven de waarde van een COBOL variabele, het bijbehorende record kan lokaliseren (met als neveneffect dat de currency indicators worden bijgewerkt). Ook de currency indicators van een settype of een recordtype kunnen met een MOVE CURRENCY STATUS worden bewaard. Het effect van deze operaties op de currency indicators wordt in 10.11.7 aan de hand van een voorbeeld toegelicht.

## 10.3 BEHANDELING VAN UITZONDERLIJKE SITUATIES

Als laatste stap in de uitvoering van een DML opdracht plaatst het DBMS een waarde in het "speciale database register" DB-STATUS om de afloop van de uitvoering van de opdracht aan te geven. Een waarde nul betekent dat de afhandeling van de opdracht succesvol is verlopen, een waarde *ongelijk aan nul* betekent dat een *fout of een uitzonderlijke situatie* is opgetreden. De COBOL gebruiker moet één of meer "declaratieve procedures" (via de USE FOR DB-EXCEPTION opdracht) aanbieden om deze gevallen te behandelen. Deze "procedures" worden bij een COBOL programma gespecificeerd aan het begin van de Procedure Division vóór de 'normale' opdrachten. We geven een voorbeeld, waarbij gebruik wordt gemaakt van een vereenvoudigde COBOL syntax.

```
   DECLARATIVES.

       USE FOR DB-EXCEPTION ON '0502100'.
EOF-PROC.
       MOVE    'YES' TO EOF.

       USE FOR DB-EXCEPTION ON '0502400'.
NOTFOUND-PROC.
       MOVE    'YES' TO NOTFOUND.

       USE FOR DB-EXCEPTION ON OTHER.
OTHER-PROC.

    END DECLARATIVES.
```

DECLARATIVES kondigt de declaratieve procedures aan. De eerste USE specificeert een declaratieve (COBOL) procedure (bestaande uit een label en de opdracht MOVE 'YES' TO EOF) die moet worden uitgevoerd als DB-STATUS door het DBMS de waarde 0502100 krijgt. Een DB-STATUS waarde bestaat uit twee delen: een aanduiding van de

opdracht, de zogenaamde 'opdrachtcode', die het zetten van DB-STATUS veroorzaakte ('05' betekent FIND) en een aanduiding van de afloop, de zogenaamde 'de status code' ('02100' betekent dat bij sequentiële toegang tot een set of realm een gewenst volgend of vorig record niet aanwezig is, zie 10.11.3). De tweede USE specificeert dezelfde procedure voor het geval DB-STATUS de waarde 0502400 ("record niet gevonden") krijgt. De derde en laatste USE specificeert een "doe-niets" (lege) procedure die moet worden uitgevoerd in die gevallen waarin DB-STATUS ongelijk aan nul is en bovendien niet de waarde 0502100 of 0502400 heeft aangenomen.

In het algemeen kan in een USE opdracht ofwel OTHER ofwel een lijst met specifieke waarden worden opgegeven. Zodra de uitvoering van een DML opdracht een waarde ongelijk aan nul in DB-STATUS veroorzaakt, zal de corresponderende declaratieve procedure aangeroepen worden. Er moet precies een (mogelijk lege) procedure zijn per waarde ongelijk aan nul. Nadat de procedure is uitgevoerd, wordt teruggekeerd naar de opdracht volgend op de DML opdracht. Het nut van een lege, declaratieve procedure is dat de programmeur de uitzonderlijke situatie 'in-line' kan afhandelen door de DML opdracht te laten volgen door een expliciete test op de DB-STATUS waarde met de juiste code om de situatie af te handelen.

Zoals gezegd kan men aan de DB-STATUS code zien welk soort opdracht de code heeft veroorzaakt: CONNECT heeft opdrachtcode 02, ERASE 04, etc. Er zijn vele statuscodes; zo betekenen bijvoorbeeld 2100 en 2300 respectievelijk 'vereist record is niet beschikbaar' en 'geen set voldoet aan de set selectiecriteria'. In [KLUW87] staat een volledige lijst met DB-STATUS codes.

## 10.4 GET

**Geef alle gegevens van leverancier L4.**

```
MOVE    'L4' TO LNO IN L
FIND    ANY L USING LNO IN L
GET     L
```

Na de uitvoering van deze opdrachten is het L record van leverancier L4 in de L locatie van de UWA gebracht. Voor het afdrukken van deze gegevens moeten COBOL opdrachten worden gegeven. Deze COBOL opdrachten zullen niet nader worden beschouwd!

Merk op dat de FIND opdracht zelf *geen* gegevenstransport bewerkstelligt. Omdat de GET (net als trouwens de meeste andere opdrachten) op de current van run-unit opereert, mag "L" weggelaten worden en kan men eenvoudig schrijven:

```
GET
```

Deze opdracht brengt de current van run-unit, van welk recordtype dan ook, in de juiste UWA lokatie. Door echter de naam van het recordtype ook te geven, stelt de programmeur het DBMS in staat om te controleren of de current van run-unit inderdaad van het juiste type is. Als in het voorbeeld de current van run-unit geen L record is, dan mislukt de GET (gevolg: DB-STATUS ongelijk aan nul!). In ieder geval is een expliciete specificatie van het recordtype duidelijker.

De meeste DML opdrachten staan toe dat de specificatie van het recordtype achterwege blijft (op dezelfde wijze als bij de GET). Het gebruik van deze mogelijkheid wordt echter niet aanbevolen en zal in de volgende voorbeelden in het algemeen niet worden gevolgd.

Het is ook mogelijk om niet het hele record, maar slechts enige velden van de current van run-unit te lezen. Dit wordt in het volgende voorbeeld gedemonstreerd.

**Geef naam en woonplaats van leverancier L4.**

```
FIND    L record van  L4
GET     LNAAM IN L, WNPL IN L
```

In dit voorbeeld worden alleen de naam en de woonplaats van leverancier L4 in de UWA gebracht; de andere UWA velden blijven ongewijzigd.

## 10.5 STORE

**Neem het LA record 'L5/A6/700' op in de database**

```
1.    MOVE    'L5' TO LNO IN LA
      MOVE    'A6' TO ANO IN LA
      MOVE     700 TO HOEV IN LA
2.    MOVE    'L5' TO LNO IN L
      MOVE    'A6' TO ANO IN A
3.    STORE    LA
```

Stap 1 bestaat uit drie MOVE opdrachten die het nieuwe LA record in de UWA construeren. Dit record kan nu in de database worden opgenomen (stap 3). LA is in het schema van Fig. 8.18 echter gedeclareerd als een AUTOMATIC member van zowel L-LA als A-LA. Dat wil zeggen dat, zodra een LA record in de database wordt opgeslagen, het DBMS dit record automatisch in de juiste sets van dit type moet invoegen. In het voorbeeld zijn dat de L-LA set van L5 en de

A-LA set van A6. In het algemeen worden de sets van type L-LA en A-LA, waar het nieuwe record moet worden ingevoegd, door het DBMS bepaald op grond van de SET SELECTION clausules zoals die in het schema voor deze settypen zijn gespecificeerd. Een volledige uitleg van deze clausule werd in paragraaf 8.8 gegeven. Om te bereiken dat het proces van SET SELECTION de juiste sets selecteert, is het in dit voorbeeld voldoende dat de programmeur de UWA velden LNO IN L en ANO IN A van adequate waarden voorziet (Waarom is dat voldoende?). Dit gebeurt in stap 2.

Zodra in het voorbeeld de STORE is uitgevoerd, is het nieuwe LA record de current van run-unit, het current LA record, het current record van realm L-LA-A en het current record van zowel L-LA als A-LA. In het algemeen wordt een zojuist opgeslagen record het current record van alle settypen waarvan het owner of AUTOMATIC member is (dit geldt niet als RETAINING gebruikt is, zie ook 10.11.7).

## 10.6 ERASE

**Verwijder het record van leverancier L4.**

```
FIND    L record van L4
ERASE   [ ALL ] L
```

Zoals het voorbeeld aantoont, zijn er twee ERASE vormen, namelijk een mèt en een zònder de optie ALL. Hun betekenis verschilt zodra het te verwijderen record (de current van run-unit) de owner is van een of meer niet-lege sets. Laten we een "afstammeling van R" recursief definiëren als zijnde een member record van een set waarvan R of een afstammeling van R de owner is. De ERASE ALL opdracht verwijdert de current van run-unit en al zijn afstammelingen; in het voorbeeld zou ERASE ALL het record van leverancier L4 en al 'zijn' LA records verwijderen. De ERASE opdracht zonder ALL daarentegen heeft een effect zoals hieronder in de stappen 1-6 is beschreven. De algemene strekking is dat de ERASE geen effect heeft als het de verwijdering van een record met MANDATORY members zou eisen; komt een dergelijk record niet voor, dan is het effect dat OPTIONAL members blijven bestaan, doch uit de set worden gehaald en dat FIXED members uit de database worden verwijderd. (Merk op dat de ERASE (zonder ALL) en de ERASE zoals die in hoofdstuk 9 is ingevoerd, gelijk zijn.)

1. Laat verwR (het te verwijderen record) de current van run-unit zijn.

2. Als er een record x bestaat dat een MANDATORY member is van een set S met owner verwR, dan mislukt de ERASE; elk "verwijderbaar" en "loskoppelbaar" merkteken dat reeds gezet is (zie de stappen 3-5 hieronder), wordt weggehaald en de ERASE

beëindigt niet succesvol (DB-STATUS ongelijk aan nul). De database blijft in precies dezelfde toestand als voor de uitvoering van de ERASE.

3. Ieder record x dat een OPTIONAL member van een set S met owner verwR is, krijgt het merkteken "loskoppelbaar van S".

4. Voor elk record x dat een FIXED member is van een set met owner verwR worden de stappen 2-5 herhaald waarbij x de rol van verwR speelt.

5. Ken het merkteken "verwijderbaar" toe aan verwR.

6. Alle records met het merkteken "loskoppelbaar" worden van de aangegeven set losgekoppeld. Alle records met het merkteken "verwijderbaar" worden losgekoppeld van alle sets waar ze deel van uitmaken (waar ze members van zijn) en daarna vernietigd. De ERASE opdracht beëindigt succesvol (DB-STATUS nul).

Merk op dat records uit de database kunnen worden verwijderd en uit sets kunnen worden losgemaakt zelfs als ze in het subschema niet "zichtbaar' zijn, dat wil zeggen, niet in het subschema zijn opgenomen.
Na een succesvolle ERASE is de currency indicator voor de "current van run-unit" gelijk aan null, waarmee 'geen positie' wordt aangeduid. Het effect op de andere currency indicators is als volgt. Noem het te verwijderen record verwR en veronderstel dat verwR het current record is van het realm M, zijn eigen recordtype, het settype S1 (waarin het owner is) en het settype S2 (waarin het member is). Stel nu dat verwR uit de database verwijderd wordt, dan worden de currency indicators als volgt bijgewerkt.

- De currency indicator van het realm M wordt zodanig bijgewerkt dat het niet langer een record aanwijst, maar het "gat" dat door verwR in M is achtergelaten. Hierdoor wordt bereikt dat bepaalde FIND opdrachten zinvol gedefinieerd kunnen worden. Zo zal een poging om het record in realm M volgend op verwR te bepalen (via een Format 3 FIND opdracht; zie paragraaf 10.11.3), resulteren in het verschuiven van de currency indicator naar het record volgend op het "gat" in M.

- De currency indicator voor het recordtype wordt op dezelfde wijze gewijzigd; dat wil zeggen dat na de ERASE de currency indicator het gat identificeert waar het record zich bevond. Een poging om het record volgend op verwR te lokaliseren door middel van een Format 1 FIND opdracht (paragraaf 10.11.1), zal resulteren in het verschuiven van de currency indicator naar het record volgend op het gat.

- De currency indicator voor het settype S1 wordt null gemaakt. FIND opdrachten zoals hiervoor aangegeven, bestaan niet voor de owner records van een settype, daarom heeft het geen zin om het begrip "gat" hier in te voeren.

- De currency indicator voor het settype S2 wordt zo gewijzigd, dat het gat wordt geïdentificeerd. Ook hier geldt weer dat de uitvoering van een opdracht (een FIND Format 3) om het record in settype S2 volgend op R te vinden, resulteert in het wijzigen van de waarde van de currency indicator van S2. Deze indicator identificeert na de wijziging het record in S2 volgend op het gat.

## 10.7 MODIFY

**Tel 10 op bij de statuswaarde van leverancier L4.**

```
FIND    L record van L4
GET     L
ADD     10 TO STATUS IN L
MODIFY  L
```

De MODIFY opdracht vervangt in de database (delen van) de current van run-unit met waarden die ontleend zijn aan de UWA. Om de waarde 10 bij de statuswaarde van L4 te tellen, moet de gebruiker eerst met een GET het record van L4 in de UWA lezen, daarna in de UWA de optelling verrichten en tenslotte het record terugschrijven (via de MODIFY). Als MODIFY de naam van een recordtype specificeert, dan moet het hele record worden vervangen. Als een lijst van velden wordt genoemd, dan moeten alleen die worden vervangen. Het gewenste effect zou in het voorbeeld ook bereikt zijn als zowel de GET als de MODIFY in plaats van L alleen het veld STATUS IN L gespecificeerd zouden hebben.

## 10.8 CONNECT

Zoals de STORE, ERASE en MODIFY opdrachten aangeboden worden om records te creëren, te vernietigen en te wijzigen, zo zijn de CONNECT, DISCONNECT en de RECONNECT opdrachten ingevoerd om het opnemen van records in sets, het halen van records uit sets en het verplaatsen van records tussen sets (van hetzelfde type) te regelen. Zodra een record member van een set is geworden, hebben we een verbinding tussen owner en member tot stand gebracht. Ter wille van het voorbeeld voeren we een nieuw settype SETX in (owner X, member L, SET SELECTION IS THRU SETX OWNER IDENTIFIED BY

APPLICATION, waarbij L zoals gewoonlijk het leverancierrecordtype is, terwijl X een willekeurig recordtype voorstelt).

**Voeg het L record van leverancier L4 toe aan de set van type SETX met owner X7.**

Het invoegen bestaat in wezen uit (1) het lokaliseren van de vereiste set van type SETX (2) het lokaliseren van het L4 record en (3) het uitvoeren van de CONNECT opdracht om de invoeging tot stand te brengen. Om stap 1 te realiseren, kan iedere willekeurige procedure gekozen zijn die ervoor zorgt dat de vereiste set van type SETX de *current set* van dat type wordt; een typerende benadering is om via een FIND de juiste owner (X7 in dit geval) te lokaliseren.

```
1.      FIND      record X7
2.      FIND      L record van L4
3.      CONNECT   L TO SETX
```

De CONNECT opdracht voegt de current van run-unit toe aan een set van het gespecificeerde type. De bedoelde set wordt bepaald in overeenstemming met de SET SELECTION criteria van het gespecificeerde settype (voor SETX is dat: 'BY APPLICATION'). Op het moment van de CONNECT mag het L record van L4 geen member van een set van type SETX zijn. Dit houdt bijvoorbeeld in dat de MEMBERSHIP CLASS van L in SETX niet INSERTION IS AUTOMATIC mag zijn, tenzij RETENTION IS OPTIONAL geldt. Merk eveneens op dat in het voorbeeld stap 1 moet worden uitgevoerd voor stap 2 (*Waarom?*). Na de CONNECT is het L record van L4 het current record van settype SETX.

## 10.9 DISCONNECT

**Haal het L record van L4 uit de SETX set waarin het zich bevindt.**

```
FIND        L record van L4
DISCONNECT  L FROM SETX
```

De DISCONNECT opdracht verbreekt het verband tussen de current van run-unit en de current set van het gespecificeerde type. Het record zelf blijft in de database bestaan, maar is niet langer een member van een set van het gespecificeerde type. Als voor de DISCONNECT de current van run-unit ook het current record van dit settype was (hetgeen niet hoeft; waarom niet?), dan wordt de currency indicator voor dat settype zo aangepast dat hij het gat in de set identificeert dat voorheen door het nu losgekoppelde record werd bezet. Let wel, de currency indicator identificeert *niet* het nog steeds bestaande record, maar de ingenomen positie. De reden voor deze aanpak is dezelfde als die bij de ERASE: bepaalde FIND opdrachten die op de members van deze set opereren, kunnen zo een zinvolle betekenis krijgen. Merk op dat L een OPTIONAL member van SETX moet zijn, anders is de DISCONNECT niet toegestaan. Merk tevens op dat het L record van L4 op het moment van de DISCONNECT een member van een zekere SETX set moet zijn.

## 10.10 RECONNECT

**Koppel het L4 record los van de huidige SETX set en voeg het toe aan de SETX set met X8 als owner.**
(Zie de voorbeelden 10.8 en 10.9.)

```
1.  FIND        record X8
2.  FIND        L record van L4
3.  RECONNECT   L WITHIN SETX
```

De RECONNECT opdracht heeft als parameter een settype (SETX in dit voorbeeld). Deze opdracht haalt de current van run-unit uit de SETX set en voegt het record toe aan een set van datzelfde type (dit mag eventueel zelfs dezelfde set zijn). Deze laatste set wordt bepaald in overeenstemming met de SET SELECTION criteria voor het gespecificeerde settype. Het effect op de currency indicators is gelijk aan die van de DISCONNECT, *behalve* als gedurende het koppelen het "gat" waar het record moet worden vastgemaakt nu net toevallig de plek is die wordt geïdentificeerd door de currency indicator van de set. In dat geval wordt de currency indicator zo aangepast dat het zojuist gekoppelde record wordt geïdentificeerd. Merk op dat in het voorbeeld L niet een FIXED member van SETX mag zijn, tenzij de SETX set waar het record wordt uitgehaald en de set waar het record wordt ingevoegd een en

dezelfde zijn (dit kan nuttig zijn als het nodig is om records in een set te verschuiven). Merk eveneens op dat de stappen 1 en 2 niet mogen worden verwisseld (net als in paragraaf 10.8).

## 10.11 FIND

Schematisch ziet een FIND opdracht er als volgt uit (zie Appendix A voor gebruikte metasymbolen):

```
FIND  record-selectie-expressie
        [ RETAINING { REALM | RECORD | {setnaam} ... }
                    CURRENCY ]
```

waarbij "record-selectie-expressie" (kortheidshalve r-s-e) een uitdrukking is die een record in de database aanwijst. De functie van FIND is dit aangewezen record te lokaliseren (dat wil zeggen, de database-sleutel ervan te bepalen) en de currency indicators bij te werken. Dit laatste betekent dat het record de current van run-unit, het current record van alle realms en alle settypen waarin het participeert, alsmede het current record van het eigen recordtype wordt gemaakt (als we de RETAINING clausule, die in 10.11.7 wordt behandeld, even vergeten). Er zijn zes algemene vormen van r-s-e, die hieronder om didactische redenen in een bepaalde volgorde worden behandeld.

**10.11.1 Toegang binnen recordtype (Format 1).** De FIND Format 1 (FIND ANY) is al enige malen geïllustreerd. Deze FIND biedt directe toegang tot een record van een bepaald type door de waarde van een (of meer) veld(en) van dat record te geven. Vaak is het veld waarvoor de waarde wordt gegeven uniek (DUPLICATES ARE NOT ALLOWED). Er bestaat echter een belangrijke variant van de FIND ANY die kan worden gebruikt als het veld (of de veldcombinatie) niet uniek is. Beide varianten worden gebruikt in het hieronder getoonde programma dat de records van de Londense leveranciers opspoort (neem aan dat er meer dan een is):

```
MOVE     'LONDEN' TO WNPL IN L
FIND     ANY L USING WNPL IN L
MOVE     'NO' TO NOTFOUND
PERFORM     UNTIL NOTFOUND = 'YES'
   GET      L
   ....
   FIND     DUPLICATE L USING WNPL IN L
END-PERFORM
```

(De PERFORM ... UNTIL definieert een lus: als de conditie achter UNTIL niet geldt, worden de opdrachten tot aan END-PERFORM

uitgevoerd; dit wordt herhaald totdat de conditie *waar* is.) Het effect van dit programma is het volgende. De eerste twee opdrachten lokaliseren *een* L record met woonplaats Londen (welk L record is afhankelijk van het DBMS, of anders gezegd is "implementor-defined"). De derde opdracht initialiseert het veld NOTFOUND tot 'NO'. De PERFORM lus leest dan het zojuist gelokaliseerde L record (via GET L), verwerkt het en probeert dan een "duplicaat" van het L record te vinden. Een duplicaat is hier een L record dat dezelfde waarde voor WNPL heeft als het current record van type L. De lus wordt herhaald totdat NOTFOUND de waarde 'YES' heeft gekregen. We nemen aan dat dit gebeurt in een declaratieve procedure "record not found" zoals die in paragraaf 10.3 is ingevoerd. Het programma zal bij uitvoering alle vereiste records vinden en zal geen records tweemaal ophalen, ook al is de volgorde waarin de records gelezen worden afhankelijk van het DBMS.

De USING clausule kan de vorm "USING identifier, identifier, ..." aannemen. In dat geval wordt gezocht naar records die de vereiste waarden voor de aangewezen velden bezitten. Het kan verhelderend werken door de toegang via een Format 1 FIND te zien als een abstractie van hashing (voor hashing, zie paragraaf 11.5).

**10.11.2 Toegang tot owner (Format 5).** Neem aan dat het current record van settype L-LA een bepaald LA record is. Vind de bijbehorende owner (dit is een L record).

```
FIND    OWNER WITHIN L-LA
```

In het algemeen bepaalt de "FIND OWNER" de owner in het gespecificeerde settype van de current set van dat type. Merk op dat het current record van de set zelf al de owner kan zijn.

**10.11.3 Sequentiële toegang in set of realm (Format 3).** Geef de ANO waarden van door L4 geleverde artikelen.

```
MOVE     'L4' TO LNO IN L
FIND     ANY L USING LNO IN L
FIND     FIRST LA WITHIN L-LA
PERFORM     UNTIL DB-STATUS = 0502100
    GET     LA
    (voeg ANO IN LA aan resultaat toe)
    FIND    NEXT LA WITHIN L-LA
END-PERFORM
```

Twee versies van de Format 3 FIND worden in dit voorbeeld getoond. De FIND FIRST lokaliseert het eerste LA record in de current set van type L-LA (namelijk de L-LA set waarvan het L4 record owner is). Dan lokaliseert, bij iedere iteratie van de lus, de FIND NEXT het volgende LA record in de current L-LA set (namelijk de L-LA set die het laatst

gevonden LA record bevat). 'Volgend' wordt geïnterpreteerd als zijnde de volgende ten opzichte van de L-LA currency indicator (dat is de positie van het LA record dat het meest recent gevonden is). We nemen (voor de verandering) aan dat er nu geen declaratieve procedure voor 0502100 beschikbaar is.

De specificatie NEXT (of FIRST) in een Format 3 FIND kan worden vervangen door PRIOR (of LAST) met een voor de hand liggende betekenis.

**10.11.4 Sequentiële toegang (met conditie) in een set (Format 6).** Geef de hoeveelheid waarmee artikel A5 door leverancier L1 kan worden geleverd.

```
MOVE        'L1' TO LNO IN L
MOVE        'A5' TO ANO IN LA
FIND        LA WITHIN L-LA USING ANO IN LA
GET         LA
(* druk de HOEV IN LA af *)
```

De derde opdracht is een Format 6 FIND. Deze opdracht lokaliseert het eerste LA record in een L-LA set dat een ANO waarde heeft gelijk aan de waarde van het veld ANO IN LA in de UWA (in dit geval 'A5'). De SET SELECTION specificatie bepaalt de af te lopen set. In Fig. 8.18 is aangegeven dat deze specificatie voor L-LA, BY VALUE OF LNO IN L is en de eerste opdracht kent de waarde 'L1' toe aan LNO IN L.

Merk op dat we in dit voorbeeld een probleem van 'strategische aard' hebben: in plaats van te beginnen met de leverancier L1 en vervolgens de set L-LA te doorlopen, hadden we ook met artikel A5 kunnen starten om daarna de set A-LA te doorlopen.

**10.11.5 Sequentiële toegang (met conditie) in een set (Format 2).** Geef alle artikelen die leverancier L1 kan leveren in de hoeveelheid 100.

```
MOVE        'L1' TO LNO IN S
MOVE        100 TO HOEV IN LA
FIND        LA WITHIN L-LA USING HOEV IN LA
PERFORM       UNTIL DB-STATUS = 0502100
    GET       LA
    ...
    FIND      DUPLICATE WITHIN
              L-LA USING HOEV IN LA
END-PERFORM
```

De eerste drie opdrachten lokaliseren het eerste LA record van leverancier L1, dat voldoet aan HOEV = 100 (dit is het LA record 'L1/A5/100'; de procedure is in essentie dezelfde als die van het vorige voorbeeld en maakt gebruik van een FIND WITHIN, dat wil zeggen een

Format 6 FIND). De volgende opdracht is duidelijk. Binnen de PERFORM lus doorloopt de FIND DUPLICATE WITHIN de current L-LA set in de NEXT richting. Er wordt gestart bij het current record van settype L-LA en gezocht naar het volgende LA record met dezelfde waarde voor HOEV als het current record. Merk op dat de Format 6 FIND hiervoor niet gebruikt kan worden omdat deze opdracht opnieuw het record 'L1/A5/100' zou lokaliseren. We hebben behoefte aan een opdracht die vanaf de current positie in voorwaartse richting de set afloopt en dat is wat de Format 2 FIND doet. De lus wordt herhaald totdat een "niet gevonden" conditie optreedt.

Het effect van de clausule 'USING HOEV IN LA' in de Format 2 FIND is, dat het DBMS zoekt naar het volgende LA record met een HOEV waarde gelijk aan die van het current record van de set. De waarde hoeft *niet* gelijk te zijn aan de waarde in het veld HOEV IN LA in de UWA. (Vergelijk deze semantiek met die van dezelfde clausule in de Format 6 FIND.) In het voorbeeld hierboven zijn genoemde twee waarden gelijk, maar de opdracht "GET LA" kan worden vervangen door bijvoorbeeld de opdracht

```
MOVE      400 TO HOEV IN LA
```

zonder dat dit effect heeft op het LA record dat door de Format 2 FIND wordt gelokaliseerd.

De USING clausule in zowel de Format 2 als de Format 6 FIND opdrachten kan de vorm "USING identifier, identifier, ..." aannemen, in welk geval er gezocht wordt naar een record dat de gewenste waarden voor alle gespecificeerde velden bevat.

**10.11.6 Toegang via database-sleutel (Format 4).** De Format 4 FIND wordt gebruikt om het record met een gespecificeerde database-sleutel waarde te "lokaliseren". Format 4 verschilt van de andere FIND's in die zin dat de *enige* functie van deze FIND is om de tabel met currency indicators bij te werken; de afhandeling van deze FIND eist geen toegang tot de database.

De "record-selectie-expressie" in een Format 4 FIND neemt de vorm aan van een *database-sleutel identifier.* Een database-sleutel identifier kan drie vormen aannemen, die we alle zullen bespreken.

**Geval 1**

Neem aan dat HULPSTR een COBOL variabele van type 'database-sleutel' is. Neem ook aan dat er een MOVE CURRENCY STATUS opdracht is uitgevoerd die een currency indicator aan HULPSTR heeft toegekend (zie 10.2). Dan zal

```
FIND      HULPSTR
```

deze 'bewaarde' database-sleutel weer current van run-unit maken (en de andere currency indicators op de normale wijze doen bijwerken). In 10.11.7 zullen we een toepassing van deze vorm zien.

**Geval 2**

We nemen aan dat het current record van L-LA een record van type LA is. Gevraagd wordt nu om het current record van het settype L-LA de current van run-unit te maken. (Merk op dat dit record niet reeds de current van run-unit hoeft te zijn.)

```
FIND      CURRENT LA WITHIN L-LA
```

Als de naam van een recordtype en een "WITHIN settype" beide zijn gespecificeerd, zoals in het voorbeeld, dan moet het current record van het aangegeven settype van het genoemde type zijn. In het voorbeeld houdt dat in, dat het current record van settype L-LA van type LA moet zijn. In plaats van een settype kan ook een realm worden aangegeven. In dat geval is het te lokaliseren record het current record van dat realm. Opnieuw geldt dat als de naam van een recordtype is genoemd, het current record van het realm van dat type moet zijn (anders krijgen we een foutmelding). Als WITHIN ... is weggelaten, terwijl de naam van het recordtype is gespecificeerd, dan is het record dat gelokaliseerd moet worden het current record van het aangegeven type. Als zowel de naam van het recordtype als WITHIN ... zijn weggelaten, dan is het "gezochte" record de current van run-unit. De algemene vorm van de "database-sleutel identifier" ziet er dan ook als volgt uit:

```
CURRENT     [ recordtypenaam ] [ WITHIN naam ]
```

waarbij "naam" de naam van een realm of een settype is.

**Geval 3**

Het derde geval berust op het begrip *keep list*. Keep lists worden in paragraaf 10.12 gedetailleerd besproken en daarom stellen we een uitvoerige discussie van deze FIND tot dan uit. Voor de volledigheid geven we echter de syntax van deze "database-sleutel identifier":

```
positie     WITHIN  keep-list-naam
```

waarbij "positie" dan wel FIRST of LAST is. Een klein voorbeeld is:

```
FIND        LAST WITHIN LISTA
```

**10.11.7 De RETAINING frase.** Stel dat de volgende vraag beantwoord moet worden: "Bepaal voor iedere leverancier die artikel A4 levert nog een artikel dat door die leverancier wordt geleverd en druk het artikelnummer alsmede het nummer en de naam van de leverancier af". Neem eenvoudigheidshalve aan, dat iedere leverancier die A4 levert ook nog minstens één ander artikel levert.

In grote lijnen ziet de vereiste procedure er als volgt uit. Beginnend met artikel A4, bekijken we ieder LA record voor dat artikel. Gegeven een LA record, bepalen we het corresponderende L record en extraheren we LNO en LNAAM. Vervolgens zoeken we in de L-LA set van de gevonden leverancier naar een LA record dat niet bij A4 hoort. Zodra we er een gevonden hebben, extraheren we de relevante ANO waarde. Gegeven de database van Fig. 10.1 zou een mogelijk resultaat van de procedure het volgende zijn (let wel, het resultaat is niet eenduidig):

| LNO | LNAAM | ANO |
|---|---|---|
| L1 | Smit | A1 |
| L4 | Klerk | A2 |

Hieronder volgt een eerste poging om het probleem in COBOL DML te formuleren.

```
1   MOVE      'A4' TO ANO IN A
2   FIND      ANY A USING ANO IN A
3   FIND      NEXT LA WITHIN A-LA
4   PERFORM     UNTIL DB-STATUS = 0502100
6      FIND     OWNER WITHIN L-LA
7      GET      L
8      MOVE     'NO' TO FOUND
9      PERFORM        UNTIL FOUND = 'YES'
10        FIND        NEXT LA WITHIN L-LA
11        GET         LA
12        IF          ANO IN LA NOT = 'A4'
13           MOVE        'YES' TO FOUND
14        END-IF
15     END-PERFORM
16     (druk LNO IN L, LNAAM IN L, ANO IN LA af)
17     FIND     NEXT LA WITHIN A-LA
18  END-PERFORM
```

(Artikel A4 is natuurlijk aanwezig. In overeenstemming met onze aanname testen we ook niet op de end-of-set conditie in de binnenste lus. Merk bovendien op dat FIND NEXT equivalent is met FIND FIRST als het current record van een settype een owner is.)

Het programma is echter niet correct; het bevat een logische fout. Probeer deze fout te vinden voordat u de uitleg hieronder leest. Het is wellicht een goed idee om het bovenstaande programma in gedachten uit te voeren op de database van Fig. 10.1. Het resultaat is

```
LNO     LNAAM       ANO
---     -----       ---
 L1      Smit        A1
 L2     Janse        A1
```

Het is interessant om op te merken dat het programma niet abnormaal eindigt. Het geeft echter een antwoord dat correct lijkt, maar het niet is.

De fout is de volgende. Als de FIND van regel 10 uitgevoerd wordt, maakt hij een LA record tot current van run-unit. Dit record, zijnde het meest recent gelokaliseerde record, wordt ook current record van alle settypen waar het deel van uitmaakt, onder andere van het A-LA settype. Dit veroorzaakt dan weer dat de A-LA set waar dit LA record deel van uit maakt, current wordt. Als bij de volgende iteratie van de buitenste lus in regel 17 de FIND NEXT wordt uitgevoerd (in een poging om de volgende leverancier van A4 te vinden), dan zal de A-LA set die in die opdracht wordt gerefereerd in het algemeen niet meer de set zijn waarvan A4 owner is. Om deze situatie te vermijden zijn er twee oplossingen voorhanden.

Ten eerste, de FIND in regel 10 kan worden uitgebreid met een daartoe geschikte RETAINING frase:

```
FIND    NEXT LA WITHIN L-LA
        RETAINING A-LA CURRENCY
```

Het doel van deze frase is om te verhinderen dat de currency indicator van settype A-LA wordt bijgewerkt. In het algemeen kan het bijwerken van currency indicators worden onderdrukt voor alle betrokken realms, voor het aan de orde zijnde recordtype, voor sommige of alle betrokken settypen, maar ook voor een willekeurige combinatie hiervan. De enige currency indicator waarvoor het bijwerken nooit kan worden onderdrukt is de "current van run-unit".
Een RETAINING frase mag worden gespecificeerd in iedere FIND of STORE opdracht.

De tweede oplossing maakt gebruik van MOVE CURRENCY STATUS en een FIND format 4. Na regel 3 en regel 17 wordt de volgende opdracht ingevoegd:

```
MOVE CURRENCY STATUS FOR A-LA TO HULPSTR
```

HULPSTR is een COBOL variabele waarin de currency indicator zolang bewaard blijft. Voor regel 17 wordt deze currency indicator weer teruggezet:

```
FIND        HULPSTR
```

De FIND NEXT LA WITHIN A-LA (regel 17) opereert nu op de goede A-LA set.

## 10.12 KEEP LISTS

**Stel dat de maximum statuswaarde van alle Parijse leveranciers nu M is. Maak voor elke Parijse leverancier de status gelijk aan M.**

Het is duidelijk dat dit probleem twee iteraties over de verzameling van Parijse leveranciers eist, namelijk een keer om M te bepalen en een keer om de wijzigingen uit te voeren. Een mogelijk programma ziet er als volgt uit.

```
MOVE        'PARIJS' TO WNPL IN L
MOVE        ZERO TO M
FIND        ANY L USING WNPL IN L
PERFORM     UNTIL DB-STATUS = 0502100
    GET         STATUS IN L
    IF          STATUS IN L > M
       MOVE     STATUS IN L TO M
    END-IF
    FIND        DUPLICATE L USING WNPL IN L
END-PERFORM

MOVE        M TO STATUS IN L
FIND        ANY L USING WNPL IN L
PERFORM     UNTIL DB-STATUS = 0502100
    MODIFY      STATUS IN L
    FIND        DUPLICATE L USING WNPL IN L
END-PERFORM
```

Aangezien in beide lussen precies dezelfde verzameling records wordt doorlopen, is het vaak veel efficiënter de adressen (database-sleutelwaarden) van de records die in de eerste lus gevonden worden, te noteren en in de tweede lus te gebruiken. Dan kan er direct naar de records worden gegaan (met behulp van de database-sleutels) in plaats van er weer naar te gaan zoeken. *Keep lists* bieden een basis voor een dergelijke benadering. Een keep list is een object, dat tot taak heeft om

een geordende lijst met database-sleutelwaarden te bewaren. Een run-unit kan elk aantal keep lists hebben. Merk op dat de keep lists geen deel uitmaken van de database, maar van de run-unit (ze zijn 'lokaal' voor de run-unit). Keep lists worden gedefinieerd in de DATA DIVISION van het COBOL programma en volgen op de opdracht die het subschema 'aanroept'. Bijvoorbeeld,

```
DB      LEVERANCIERS WITHIN LEVERANCIERS-ARTIKELEN.
LD      LISTA LIMIT IS 15.
LD      LISTB LIMIT IS 20.
```

Gedurende de uitvoering van dit programma zullen de keep lists LISTA en LISTB beide een geordende lijst van database-sleutelwaarden bevatten en op deze wijze ieder een verzameling records in de database aanwijzen. LISTA kan maximaal 15 en LISTB maximaal 20 waarden bevatten. Het plaatsen van een database-sleutelwaarde in een keep list geschiedt via een KEEP opdracht, terwijl verwijdering uit een keep list gebeurt met een FREE opdracht. Deze opdrachten worden later meer in detail besproken.

Terugkerend naar het onder handen zijnde probleem tonen we een oplossing met gebruikmaking van één keep list, namelijk LISTA zoals hierboven gedefinieerd (we nemen aan dat er niet meer dan 15 Parijse leveranciers zijn).

```
MOVE       'PARIJS' TO WNPL IN L
MOVE       ZERO TO M
FIND       ANY L USING WNPL IN L
PERFORM    UNTIL DB-STATUS = 0502100
     KEEP       USING LISTA
     GET        STATUS IN L
     IF         STATUS IN L > M
        MOVE    STATUS IN L TO M
     END-IF
     FIND       DUPLICATE L USING WNPL IN L
END-PERFORM

MOVE       M TO STATUS IN L
MOVE       'NO' TO LISTEMPTY
PERFORM    UNTIL LISTEMPTY = 'YES'
     FIND       FIRST WITHIN LISTA
     MODIFY     STATUS IN L
     FREE       FIRST WITHIN LISTA
END-PERFORM
```

Elke keer als de KEEP opdracht wordt uitgevoerd, voegt het de database-sleutelwaarde van de current van run-unit (hier: het record van

een Parijse leverancier) toe aan het eind van de gespecificeerde keep list (LISTA). Bij het verlaten van de eerste lus bevat LISTA daarom de database-sleutelwaarden van alle Parijse leveranciers en wel in de volgorde waarin ze werden gevonden. De tweede lus doorloopt dan de verzameling records die door deze database-sleutels worden aangewezen. Bij iedere stap wordt het record, bepaald door de eerste waarde van de list, benaderd en daarna wordt deze waarde van de lijst verwijderd. (Merk echter op dat de beëindigingsconditie van de lus niet "record not found" maar "keep list empty" is. We tonen hier niet de details voor het behandelen van deze conditie, maar nemen aan dat het normale patroon met declaratieve procedures gevolgd wordt.) Zoals we in 10.11.6 al aangaven, is de expressie FIRST WITHIN LISTA een "database-sleutel identifier". De algemene vorm van FREE is

```
FREE      database-sleutel-identifier
```

of

```
FREE      ALL  [ FROM { keep-list-naam } ... ]
```

Als in de eerste vorm de database-sleutel identifier een van de keep list vormen is (FIRST of LAST), dan wordt de aangegeven database-sleutel waarde uit de genoemde keep list gehaald. In de tweede vorm worden alle waarden uit de aangegeven keep lists (of, als de FROM frase ontbreekt, uit alle keep lists) verwijderd; met andere woorden, de genoemde keep lists worden leeg (of, als de FROM frase ontbreekt, álle keep lists worden leeg).

## 10.13 READY EN FINISH, COMMIT EN ROLLBACK

We besluiten dit hoofdstuk met een korte beschrijving van de overblijvende DML opdrachten (READY en FINISH; COMMIT en ROLLBACK).

**10.13.1 READY en FINISH.** De READY opdracht maakt een realm beschikbaar voor verwerking, bijvoorbeeld,

```
READY     L-LA-A USAGE-MODE IS UPDATE
```

USAGE-MODE is òf RETRIEVAL òf UPDATE. Ze mogen verder worden gekwalificeerd als PROTECTED of EXCLUSIVE, waarmee de eisen van de run-unit met betrekking tot het gelijktijdig opereren van andere run-units op hetzelfde realm worden aangegeven (het is dan de taak van het DBMS om deze eisen waar te maken). PROTECTED betekent dat het realm gelijktijdig kan worden gebruikt met run-units die alleen leesoperaties uitvoeren, terwijl dit niet geldt voor run-units die ook

schrijven. EXCLUSIVE houdt in dat de run-unit het realm voor zich alleen moet hebben.

FINISH maakt een beschikbaar realm niet beschikbaar, geeft het realm als het ware aan het DBMS terug. Zie bijvoorbeeld de volgende opdracht:

```
FINISH      L-LA-A
```

**10.13.2 COMMIT en ROLLBACK.** Bij de uitleg van de werking van deze twee opdrachten kan men zich het beste voorstellen, dat het DBMS voor iedere run-unit een bij deze run-unit horende werkruimte heeft waar de wijzigingen die door de run-unit in de database worden aangebracht zo lang worden bewaard. Andere run-units zien nog niets van deze (voorlopige) wijzigingen.

De COMMIT opdracht creëert een zogenaamd "rustpunt" voor de run-unit (het begin van een run-unit is ook zo'n rustpunt). Alle veranderingen die door de run-unit in zijn werkruimte zijn aangebracht, worden door het DBMS naar de database geschreven. Deze veranderingen zijn dan definitief en voor andere run-units zichtbaar geworden. Eenmaal in de database opgenomen, kunnen ze niet meer (via een ROLLBACK, zie hieronder) ongedaan gemaakt worden. De laatste opdracht van een run-unit met bijwerkoperaties is normaal gesproken een COMMIT. Zonder deze laatste COMMIT zouden alle wijzigingen sinds de voorlaatste COMMIT verloren gaan, omdat ter afsluiting van een run-unit het DBMS *automatisch* genoemde werkruimte leegt. (Heel vaak, maar niet altijd, zal de eventuele COMMIT op het eind van de run-unit ook de enige zijn!).

De uitvoering van de ROLLBACK opdracht leegt genoemde werkruimte en doet dus alle (voorlopige) wijzigingen sinds de laatste COMMIT teniet. De run-unit kan deze ROLLBACK zinvol gebruiken als bijvoorbeeld in het midden van de verwerking van een grote hoeveelheid invoer een fout ontdekt wordt: alle tot nu toe aangebrachte wijzigingen zijn dan waardeloos geworden en moeten ongedaan gemaakt worden. Merk op dat de run-unit na de ROLLBACK nog steeds in uitvoering is en dus nog bijwerkingen kan plegen die eventueel wèl met een COMMIT kunnen worden afgesloten.

## 10.14 DATABASE-CONDITIES

Database-condities stellen het (COBOL) programma in staat om condities te formuleren met behulp van de huidige toestand van de database. De database-condities die behandeld zullen worden, zijn de zogenaamde TENANT en MEMBER condities.

**10.14.1 TENANT conditie.** Deze conditie bepaalt of de current van run-unit deel uitmaakt van een zeker settype. Dit kan zijn als member (via het sleutelwoord MEMBER), als owner (via OWNER) of als member of owner (via TENANT). In onderstaand voorbeeld wordt aangenomen dat A4 door tenminste één leverancier geleverd wordt.

```
MOVE        'A4' TO ANO IN A
FIND        ANY A USING ANO IN A
FIND        NEXT LA WITHIN A-LA
IF          MEMBER OF L-LA
.....
```

De afhandeling van MEMBER OF L-LA houdt in dat gekeken wordt of het current record van run-unit member is van een set van type L-LA. (Aangezien de current van run-unit van type LA is, zal dit in de leveranciers-artikelen database altijd het geval zijn; weet u waarom?.) De evaluatie van de conditie OWNER OF L-LA resulteert in *waar* als de current van run-unit owner is in een set van type L-LA. Als TENANT OF L-LA is gespecificeerd, dan resulteert de test in *waar* als de current van run-unit òf owner òf member van een L-LA set is.

**10.14.2 MEMBER conditie.** Met deze conditie is te bepalen of een zekere set leeg is of niet. (Het is enigszins verwarrend, dat het sleutelwoord MEMBER voorkomt in de TENANT conditie, zie 10.14.1, en niet in de MEMBER conditie.) Afhankelijk van de test kan het (COBOL) programma dan zekere acties ondernemen. De test van de conditie

```
L-LA IS EMPTY
```

resulteert in *waar* als de current set van type L-LA leeg is. Anders is het resultaat vanzelfsprekend *onwaar*. Men kan natuurlijk ook op een andere wijze ontdekken of de current L-LA set leeg is of niet (*Vraag*: hoe?), maar dat gaat wat omslachtiger. Als het programma bijvoorbeeld alle leveranciers moet opsporen die geen artikelen kunnen leveren, dan kan de MEMBER conditie zinvol gebruikt worden. Door toevoeging van NOT kan gekeken worden of de current L-LA set niet leeg is:

```
L-LA IS NOT EMPTY
```

## OPGAVEN

**10.1** Gegeven is Fig. 10.1 en het schema van Fig. 8.18. Beschouw de volgende reeks opdrachten:

```
MOVE        'L2' TO LNO IN L
FIND        ANY L USING LNO IN L
FIND        FIRST LA WITHIN L-LA
                RETAINING LA CURRENCY
PERFORM     UNTIL DB-STATUS = 0502100
    FIND        NEXT LA WITHIN L-LA
END-PERFORM
```

Stel de tabel met currency indicators op die ontstaan is na de uitvoering van ieder van de FIND opdrachten.

**10.2** Gebruik bij de volgende vragen uw antwoord op opgave 8.5 (het schema voor de leveranciers-artikelen-projecten database). Formuleer DBTG oplossingen bij de volgende vragen:

a. Geef de LNO waarden van die leveranciers die aan project P1 leveren.
b. Geef de LNO waarden van die leveranciers die artikel A1 leveren aan project P1.
c. Geef de LNO waarden van die leveranciers die project P1 voorzien van een rood artikel.
d. Geef de PNO waarden van die projecten waaraan niet een rood artikel wordt geleverd door een Londense leverancier.
e. Geef de ANO waarden van die artikelen die geleverd worden door tenminste een leverancier die tenminste een artikel levert dat ook door leverancier L1 wordt geleverd.
f. Geef alle <WNPL,STAD> paren zodat een leverancier in WNPL aan een project levert in STAD.
g. Verander de kleur van alle rode artikelen in oranje.
h. De hoeveelheid van A1 die door L1 aan P1 wordt geleverd, wordt nu door L2 geleverd. Breng de noodzakelijke veranderingen aan.

**10.3** Beschouw het 'artikelen' schema van Opgave 8.6. Breid dit schema uit met een singular settype ARTIKEL-SET dat alle artikelen bevat.

a. Geef voor elk artikel het artikelnummer en de nummers van onmiddellijke componenten ervan, alsmede de artikelnummers van de artikelen waarvan het een onmiddellijke component is.
b. Geef voor een gegeven artikel de nummers van alle componenten van welk niveau dan ook (het materiaalexplosie probleem).

# 11
# Het Interne Niveau

## 11.1 INLEIDING

Het interne niveau van een database systeem houdt zich bezig met het opslaan van gegevens op externe geheugens (schijf, magneetband, etc), zie ook paragraaf 2.2. Op dit niveau ligt de nadruk op de *efficiënte* afhandeling van de lees- en bijwerkoperaties die ten behoeve van de toepassingen op de database worden uitgevoerd. Als criterium geldt vooral het minimaliseren van de tijd die vereist is voor deze operaties. Deze tijd wordt sterk beïnvloed door de manier waarop gegevens zijn opgeslagen. Een tweede doel dat vaak nagestreefd wordt, is het minimaliseren van de ruimte die de database op het externe geheugen in beslag neemt.

Dit hoofdstuk poogt een inzicht te geven in de diverse opslagtechnieken die zoal gebruikt worden.

Een gegeven rangschikking van gegevens op een extern geheugen noemen we een *opslagstructuur*. Er kunnen vele opslagstructuren worden ontworpen, elk met zijn eigen prestatie-karakteristieken. Een opslagstructuur kan een snelle verwerking van de ene toepassing toestaan, terwijl de verwerking van een andere juist lang duurt. Meestal gebruikt een toepassing niet de hele database; aangezien een database wordt 'gedeeld' (zie voor deze term paragraaf 1.4) kan een bepaald deel best door vele toepassingen worden gebruikt. Het proces om te komen tot de bepaling van de beste database-opslag heet *fysiek database-ontwerp*. (Zie voor meer informatie over dit onderwerp [TEFR82].) De zojuist genoemde aspecten maken het fysiek database-ontwerp tot een complex proces.

**Fysiek database-ontwerp.** Het is zeer onwaarschijnlijk dat één opslagstructuur optimaal is voor de opslag van alle delen van een database. Dat betekent dat een goed DBMS vele opslagstructuren moet

aanbieden, zodat bij elk deel van de database een 'optimale' opslagstructuur gekozen kan worden. Fysiek database-ontwerp is voor grote en complexe databases een moeilijke taak. Het vereist (onder andere) een goed begrip van de toepassingen die van de database gebruik maken en van de frequentie waarmee die toepassingen worden uitgevoerd. Daarnaast moeten de eigenschappen van de externe geheugens, waarop de database moet worden geplaatst, goed bekend zijn. Aangezien databases bijna zonder uitzondering op direct-toegankelijke geheugens worden opgeslagen en dan nog speciaal op schijven, gaan we wat uitvoeriger op schijven in. We nemen ter wille van de eenvoud (en zonder verlies van algemeenheid) aan dat de hele database op één schijvenpakket kan worden opgeslagen.

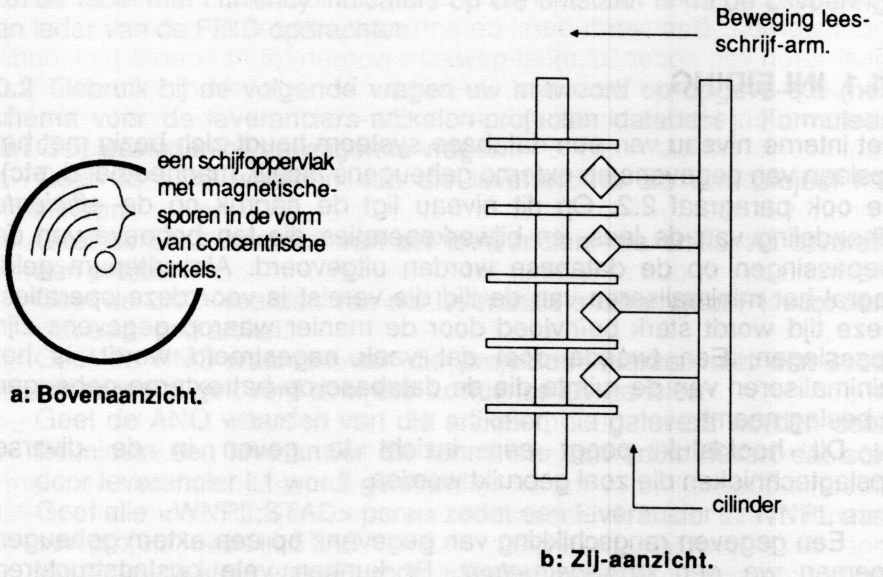



**Fig. 11.1 Een schijvenpakket (Schematisch).**

**Schijven.** In Fig. 11.1 wordt schematisch een schijvenpakket met vijf schijven getekend. De schijven zijn boven elkaar geplaatst en iedere schijf heeft een boven- en onderoppervlak; alle oppervlakken, behalve de bovenkant van de bovenste en de onderkant van de onderste, zijn in het algemeen beschrijfbaar en 'in gebruik'. Op een oppervlak zijn magnetische sporen in de vorm van concentrische cirkels aangebracht (zie Fig. 11.1.a). Op deze sporen worden blokken (of fysieke records) geschreven; een blok is de eenheid van lezen of schrijven. In Fig. 11.1.b

wordt een zij-aanzicht van het schijvenpakket getoond. De sporen op de diverse oppervlakken met gelijke straal vormen tezamen een cilinder. Een schijvenpakket heeft evenveel cilinders als er sporen op een oppervlak van een schijf zitten. Per cilinder worden de sporen die er deel van uitmaken genummerd 0, 1, 2, ... Een blok heeft een *fysiek schijfadres* bestaande uit het cilindernummer, het spoornummer en het bloknummer; adres 19-02-03 betekent bijvoorbeeld blok 03 van spoor 02 op cilinder 19.

Als een schijveneenheid in bedrijf is, dan roteren de schijven; de tijd nodig om een omwenteling te realiseren, wordt de rotational delay genoemd (een rotational delay van 25 milliseconde is normaal). Iedere schijveneenheid heeft een beweegbare lees/schrijfarm, die per schijfoppervlak een lees/schrijfkop bezit. Stel dat blok 19-02-03 gelezen moet worden. Dan wordt eerst de arm naar cilinder 19 gebracht. Na het lokaliseren van spoor 02 moet gewacht worden tot het derde blok onder de lees-schrijfkop verschijnt, waarna het *gegevenstransport* kan plaatsvinden. Het verplaatsen van de arm wordt een *seek* genoemd. De gemiddelde tijd die nodig is voor een seek ligt (afhankelijk van het type schijf) tussen enige tientallen en enige honderden milliseconden. Het vinden van het goede spoor wordt in prestatiebeschouwingen altijd verwaarloosd, omdat dit in tegenstelling tot de armverplaatsing electronisch (en dus zeer snel) kan gebeuren. Het zoeken naar het juiste blok op een spoor heet de *search-operatie*.

Belangrijk hierbij is dat de toegang tot gegevens op een schijf veel langer duurt dan toegang tot het hoofdgeheugen: het verschil is een factor 10.000 tot 100.000. Ondanks het feit dat het hoofdgeheugen veel vaker benaderd moet worden dan het schijfgeheugen, is de belangrijkste prestatie-doelstelling in database systemen meestal het minimaliseren van het aantal lees/schrijfoperaties van/naar schijf. Dit hoofdstuk zal zich met deze problematiek bezighouden, dat wil zeggen, er zullen technieken worden besproken die het mogelijk maken de gewenste gegevens (bijvoorbeeld een opgeslagen record) in zo min mogelijk bloktoegangen op te halen.

**Bestanden.** Op het conceptuele niveau hebben we te maken met tupels of records, afhankelijk van het data model waarin het conceptuele schema is uitgedrukt. Op het interne niveau zullen we over opgeslagen records spreken. We definiëren een *opgeslagen bestand* als de verzameling van alle opgeslagen records van een zeker type. De term opgeslagen record behoeft enige uitleg.

De afbeelding van bijvoorbeeld een tupel uit het relationele model naar opgeslagen records is niet eenduidig. Stel dat in de L relatie (van de leveranciers-artikelen database) de attribuutwaarden van LNO, LNAAM en STATUS veel en WNPL weinig gelezen worden. Neem bovendien aan dat namen van woonplaatsen erg lang zijn (stel enige

honderden karakters). Dan zou de database administrator (DBA) kunnen overwegen om de L relatie in twee bestanden op te slaan: één bestand met opgeslagen records bestaande uit de attributen LNO, LNAAM en STATUS, en één met opgeslagen records die de attributen LNO en WNPL bevatten. Als bovendien het eerste bestand op een (extra) snel medium zou worden geplaatst, dan zou het lezen van het eerste bestand drastisch worden versneld. Aangezien dat bestand veel gelezen moet worden, zal de totale prestatie van het database systeem hierdoor toenemen.

Na deze inleiding tenslotte iets over de inhoud van dit hoofdstuk. In paragraaf 11.2 wordt een globaal overzicht van het lokaliseren en benaderen van een bepaald record in de database gegeven. Bovendien worden de belangrijkste programmatuur-componenten genoemd die daarbij een rol spelen. Paragraaf 11.3 behandelt twee van die componenten in wat meer detail, namelijk de bestand- en de schijfbeheerder. De paragrafen 11.2 en 11.3 kunnen eventueel bij eerste lezing overgeslagen worden omdat de beschreven details niet direct nodig zijn om de rest van het hoofdstuk te begrijpen. De paragrafen 11.4-11.6 bevatten het belangrijkste deel van dit hoofdstuk, namelijk bekende opslagstructuren zoals "indexering", "hashing" en "wijzerketens". Van deze opslagstructuren zijn vele varianten bekend, elk met voor- en nadelen. Het voert echter te ver om al die varianten te behandelen, vandaar dat de nadruk meer zal liggen op concepten dan op details. Tenslotte geeft paragraaf 11.7 enige slotopmerkingen. Bij het beschrijven van opslagstructuren worden voorbeelden gebruikt; deze voorbeelden zijn (weer) gebaseerd op de leveranciers-artikelen database, waarvan Fig. 7.1. de inhoud geeft.

## 11.2 TOEGANG TOT DE DATABASE: EEN OVERZICHT

Voordat we op de specifieke opslagstructuren ingaan, schetsen we eerst hoe toegang tot een database kan worden verkregen. Het lokaliseren en ophalen van een gegeven in de database en het presenteren van dit gegeven aan de gebruiker worden uitgevoerd door vele lagen van systeemprogrammatuur. Vanzelfsprekend zullen de details van systeem tot systeem verschillen (dit geldt zelfs voor de terminologie), maar toch zijn de principes redelijk standaard. Een en ander zal aan de hand van Fig. 11.2 worden uitgelegd.

1. Het *DBMS* bepaalt welk opgeslagen record vereist is en vraagt aan de *bestandbeheerder* om dat record op te halen. (We nemen in dit verband aan, dat het DBMS in staat is om het vereiste record precies aan te wijzen. In de praktijk is dat soms niet mogelijk en kan alleen een verzameling gedefinieerd worden waar het gezochte record deel van uit moet maken. Dit kan inhouden dat de stappen

1-3 een aantal keren moeten worden uitgevoerd. Zodra de records in het hoofdgeheugen zijn gebracht, kan daar het juiste record worden bepaald.)

2. Op zijn beurt bepaalt de bestandbeheerder welke pagina het gezochte record bevat en vraagt de *schijfbeheerder* die pagina op te halen. Een pagina is de eenheid van transport tussen bestand- en schijfbeheerder. Pagina's zijn van vaste lengte; veel voorkomende paginagroottes zijn 1024, 2048 en 4096 bytes.

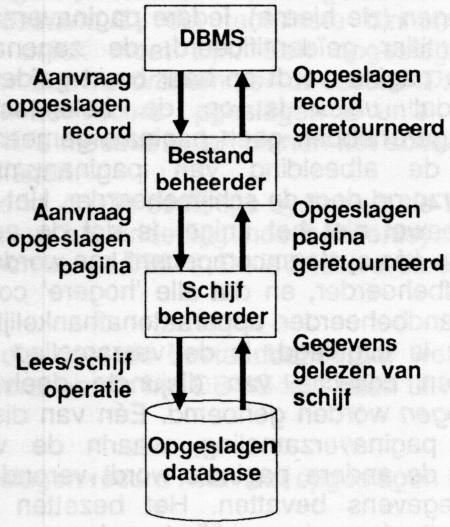



**Fig. 11.2 Het DBMS, de bestand- en de schijfbeheerder.**

3. Tenslotte bepaalt de schijfbeheerder in welk blok de gewenste pagina zich bevindt. (In dit hoofdstuk wordt, zonder verlies van algemeenheid, aangenomen dat een blok precies één pagina kan bevatten en dat de vertaling van paginanummer naar fysiek schijfadres dus eenvoudig is.) Hierna geeft de schijfbeheerder de noodzakelijke schijfopdrachten om het blok (en dus de pagina) te lezen.

Globaal gesproken ziet het DBMS daarom de database als een verzameling opgeslagen records, een visie die door de bestandbeheerder wordt mogelijk gemaakt. De bestandbeheerder daarentegen ziet de database als een verzameling pagina's en die visie wordt door de schijfbeheerder mogelijk gemaakt. De schijfbeheerder

tenslotte, ziet de database zoals hij 'echt' op schijf is opgeslagen. Hieronder wordt een en ander nader uitgewerkt.

**11.2.1 Schijfbeheerder.** De schijfbeheerder is een deel van het onderliggende beheerssysteem en is verantwoordelijk voor alle fysieke in- en uitvoeroperaties. Als zodanig moet de schijfbeheerder zich bewust zijn van het fysieke schijfadres. Als de bestandbeheerder bijvoorbeeld vraagt om pagina p op te halen, dan moet de schijfbeheerder precies weten waar deze pagina zich op schijf bevindt. De gebruiker van de schijfbeheerder (dat is de bestandbeheerder) behoeft dat echter niet te weten. Deze bestandbeheerder ziet de schijf als een collectie van paginaverzamelingen (zie hierna). Iedere paginaverzameling wordt door een unieke identifier geïdentificeerd, de zogenaamde *paginaverzameling-ID*. Elke pagina wordt op haar beurt geïdentificeerd door een *paginanummer* dat *uniek* is op de hele schijf; verschillende paginaverzamelingen hebben geen pagina's gemeenschappelijk. Zoals gezegd, wordt de afbeelding van paginanummers op fysieke schijfadressen verzorgd door de schijfbeheerder. Het grote voordeel van deze aanpak (hoewel niet het enige) is dat de apparaatafhankelijke programmacode in één systeemcomponent kan worden samengebracht, namelijk de schijfbeheerder, en dat alle 'hogere' componenten, in het bijzonder de bestandbeheerder, apparaatonafhankelijk kunnen blijven.

Zoals zojuist is uitgelegd, is de verzameling pagina's op schijf opgedeeld in een collectie van disjuncte deelverzamelingen, die *paginaverzamelingen* worden genoemd. Eén van die verzamelingen is de *vrije ruimte* paginaverzameling, waarin de vrije pagina's zijn opgenomen; van de andere pagina's wordt verondersteld dat ze van belang zijnde gegevens bevatten. Het bezetten en vrijgeven van pagina's geschiedt door de schijfbeheerder op aanvraag van de bestandbeheerder. De operaties op de paginaverzamelingen die door de schijfbeheerder worden ondersteund (en die dus ter beschikking van de bestandbeheerder staan), zijn onder andere:

- Lees pagina p van paginaverzameling V.
- Vervang pagina p in paginaverzameling V.
- Voeg een nieuwe pagina toe aan V (dat wil zeggen, verkrijg een lege pagina p van de vrije ruimte paginaverzameling, voeg p toe aan V en geef het paginanummer p aan de aanvrager).
- Verwijder pagina p van V.

De eerste twee operaties zijn natuurlijk de basisoperaties; de andere twee maken het mogelijk dat een paginaverzameling groeit en krimpt.

**11.2.2 Bestandbeheerder.** De bestandbeheerder gebruikt de zojuist beschreven faciliteiten van de schijfbeheerder op een zodanige wijze, dat de gebruiker van de bestandbeheerder (het DBMS) de schijf kan zien als een collectie opgeslagen records in opgeslagen bestanden. Iedere paginaverzameling zal één of meer opgeslagen bestanden bevatten. We zullen verderop (bij groepering) zien dat het DBMS zich bewust moet zijn van paginaverzamelingen, ook al is het niet verantwoordelijk voor het beheer ervan. In het bijzonder moet het DBMS weten, welke bestanden in dezelfde paginaverzameling en welke records op dezelfde pagina zijn opgeslagen.

Ieder opgeslagen bestand wordt geïdentificeerd door een *bestandnaam* of *bestand-ID*, die tenminste binnen de omvattende paginaverzameling uniek moet zijn. Een opgeslagen record wordt geïdentificeerd door een *recordnummer* of *record-ID*, die tenminste in het bestand waarin het record opgeslagen is uniek moet zijn. In de praktijk zijn record-ID's meestal uniek binnen het schijvenpakket waarop het record is opgeslagen.

Het zij opgemerkt, dat in sommige systemen de bestandbeheerder een component is van het onderliggende beheerssysteem en dat hij in andere is opgenomen in het DBMS. Dit is in de verdere uiteenzetting niet van belang.

De operaties die door de bestandbeheerder worden ondersteund (dat zijn dus de operaties die het DBMS kan laten uitvoeren), zijn onder andere:

- Lees het opgeslagen record r van het opgeslagen bestand b.
- Vervang het opgeslagen record r in het opgeslagen bestand b.
- Voeg een opgeslagen record toe aan het opgeslagen bestand b en geeft de nieuwe record-ID r terug.
- Verwijder opgeslagen record r van het opgeslagen bestand b.
- Creëer een nieuw opgeslagen bestand b.
- Vernietig het opgeslagen bestand b.

Met deze operaties ten behoeve van het bestandbeheer is het DBMS in staat om de door haar gewenste opslagstructuren te bouwen en te onderhouden. Deze opslagstructuren vormen, zoals bekend, het belangrijkste onderwerp van dit hoofdstuk (zie paragrafen 11.4-11.7).

**11.2.3 Groepering (clustering).** We kunnen dit overzicht niet beëindigen zonder kort het onderwerp *groepering* (clustering) aan te snijden. Het idee bij groepering is om records die logisch gerelateerd zijn

(en daarom vaak tezamen gebruikt worden) ook fysiek dicht bij elkaar op te slaan. Groepering speelt een grote rol bij het behalen van snelheid in het afhandelen van met name vragen (en dat is, zoals bekend, op het interne niveau een belangrijke doelstelling). Een eenvoudig voorbeeld zal dit illustreren. Stel dat het record dat het laatst gelezen is het record r1 is en veronderstel dat het volgende record dat gelezen moet worden r2 is. Stel nu dat r1 op pagina p1 en r2 op pagina p2 is opgeslagen.
Dan geldt:

1. Als p1 en p2 gelijk zijn, dan hoeft r2 niet afzonderlijk opgehaald te worden, omdat de vereiste pagina p2 al ingelezen is.

2. Als p1 en p2 verschillend zijn, dan zal het lezen van r2 (en dus van p2) een toegang tot de schijf vereisen. Als p1 en p2 dicht bij elkaar op schijf staan, dan zal de seektijd echter klein zijn (en zelfs nul als p1 en p2 op dezelfde cilinder zijn opgeslagen), omdat de lees/schrijfkoppen al dicht bij de gewenste plaats staan.

Ter verdere illustratie beschouwen we de leveranciers-artikelen database:

- Als de leverancierrecords vaak gesorteerd op nummer moeten worden getoond, dan is het efficiënt om het record van L1 naast het record van L2, en het record van L2 naast dat van L3, etc. op te slaan. Dit is een voorbeeld van *intra-bestand* groepering: de groepering wordt toegepast op slechts één bestand. (Let wel: "intra" betekent "binnen".)

- Als, aan de andere kant, vaak de gegevens van een leverancier met de door hem leverbare artikelen vereist zijn, dan zal een leverancierrecord gevolgd moeten worden door de records die de leverbare artikelen van deze leverancier beschrijven. Dat is een voorbeeld van *inter-bestand* groepering: de groepering heeft betrekking op records van meer dan een bestand ("inter" betekent "tussen".)

Vanzelfsprekend kan een bestand slechts op één manier fysiek gegroepeerd worden.

Het DBMS kan inter- en intrabestand groepering ondersteunen door logisch gerelateerde records fysiek dicht bij elkaar op te slaan. Daarom moet het DBMS de begrippen opgeslagen bestand en paginaverzameling kennen. Als het DBMS een record creëert, dan moet het aan de bestandbeheerder doorgeven in de "buurt" van welk ander, reeds opgeslagen record het nieuwe record moet worden opgeslagen. Het is de taak van de bestandbeheerder en de schijfbeheerder om dat dan te realiseren. Zo wordt bereikt dat het DBMS groepering kan

afdwingen, terwijl de implementatie-aspecten toch door de onderliggende lagen worden afgehandeld.

Natuurlijk weet het DBMS alleen welke groepering gewenst is als de database administrator dat duidelijk heeft gemaakt. Een goed DBMS staat de DBA toe om verschillende soorten groepering te definiëren voor verschillende bestanden. Het moet ook mogelijk zijn om de groepering van een bestand te wijzigen zonder dat de programma's hierdoor aangepast moeten worden (fysieke gegevensonafhankelijkheid, hoofdstuk 1).

Het is van belang om nogmaals op te merken dat een goed fysiek database-ontwerp zonder kennis van de toepassingen onmogelijk is.

## 11.3 PAGINAVERZAMELINGEN EN BESTANDEN

Een van de belangrijkste functies van de schijfbeheerder is het de bestandbeheerder mogelijk maken om alle details ten aanzien van fysieke in- en uitvoeroperaties van en naar schijf te negeren (zie ook de vorige paragraaf). Deze functie wordt *paginabeheer* genoemd. We geven een heel eenvoudig voorbeeld om de principes van paginabeheer uit te leggen. Beschouw hiertoe nog eens de leveranciers-artikelen database. Veronderstel dat het gewenst is dat de records van leveranciers op leveranciernummer en die van artikelen op artikelnummer gesorteerd zijn. Neem bovendien aan dat records van leverbare artikelen geordend moeten zijn op artikelnummer binnen leveranciernummer. Eenvoudigheidshalve veronderstellen we, dat ieder opgeslagen bestand in één paginaverzameling is opgeslagen en dat elk opgeslagen record precies één pagina beslaat (dit laatste is niet echt realistisch). Beschouw nu de volgende gebeurtenissen.

1. In het begin bevat de database geen records. Er is dan één paginaverzameling, namelijk de 'vrije ruimte' paginaverzameling, die alle pagina's van de schijf behalve pagina nul (zie later), bevat. De lege pagina's zijn sequentieel genummerd vanaf één.

2. De bestandbeheerder vraagt de creatie van een paginaverzameling voor de leverancierrecords en voegt daar vijf records (voor L1-L5) aan toe. De schijfbeheerder verwijdert de pagina's 1-5 van de 'vrije ruimte' paginaverzameling en voegt ze toe aan de paginaverzameling van de leveranciers.

3. Voor artikelen en leverbare artikelen krijgen we iets dergelijks. Nu zijn er vier paginaverzamelingen (*Welke*?), zie Fig. 11.3. (Merk op dat op pagina 12 het LA record met sleutel L1A1 is opgeslagen. Dus niet het L record L1 en het A record A1).

Vervolgens komen de stappen:

4. De bestandbeheerder voegt een nieuw record (voor leverancier L6) toe. De schijfbeheerder lokaliseert de eerste vrije pagina in de paginaverzameling van de vrije ruimte (dat is pagina 24) en voegt deze pagina toe aan de paginaverzameling van de leveranciers.

5. De bestandbeheerder verwijdert het opgeslagen record van leverancier L2. De schijfbeheerder geeft de pagina van L2 (pagina 2) terug aan de paginaverzameling van de vrije ruimte.

6. De bestandbeheerder voegt een nieuw artikel record (voor artikel A7) toe. De schijfbeheerder lokaliseert de eerste vrije pagina (pagina 2) en voegt die toe aan de paginaverzameling van de artikelen.

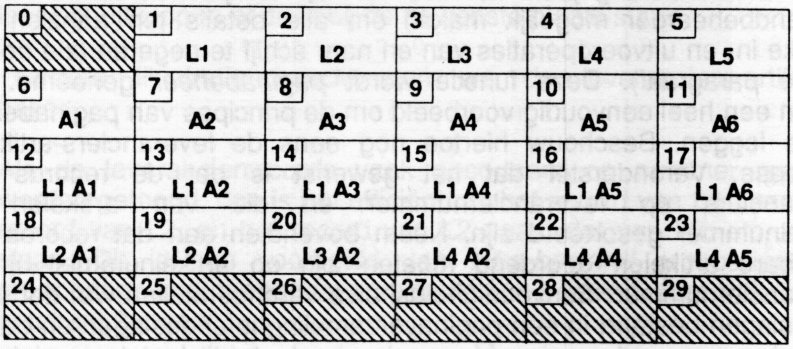



**Fig. 11.3 Layout van schijf na het laden van de leveranciers-artikelen database.**

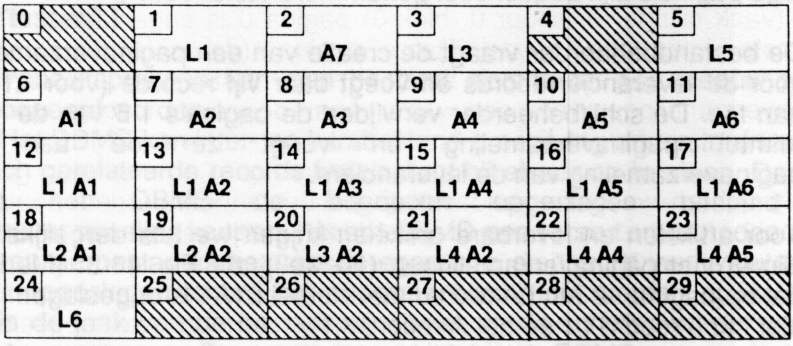



**Fig. 11.4 Layout van schijf na enige bijwerkoperaties.**

7. De bestandbeheerder verwijdert het opgeslagen record van leverancier L4. De schijfbeheerder geeft de bijbehorende pagina (pagina 4) terug aan de vrije ruimte.

De verkregen situatie is als geschetst in Fig. 11.4. Deze figuur illustreert zeer duidelijk het volgende: nadat het systeem een tijdje heeft gewerkt, liggen pagina's die logisch bij elkaar horen (bijvoorbeeld de pagina's van de leverancierrecords) fysiek *niet* meer bij elkaar, ook al was dat in de beginsituatie wel zo. Daarom moet de logische volgorde van pagina's niet door een fysieke volgorde, maar door wijzers worden vastgelegd. Daartoe krijgt iedere pagina een *pagina-kop*, waarin onder andere het paginanummer van de (logisch gezien) volgende pagina staat. zie Fig. 11.5.

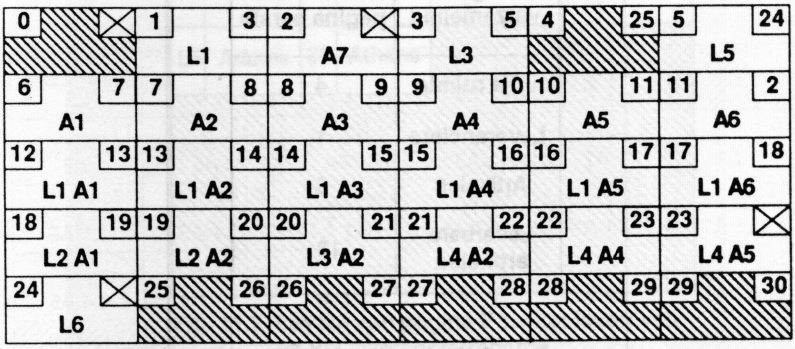



**Fig. 11.5 Figuur 11.4 herzien om de "volgende pagina" wijzers te tonen. (Rechterbovenhoek van elke pagina.)**

Hoe weet de schijfbeheerder waar de diverse paginaverzamelingen beginnen? (Het is voldoende om de eerste pagina van een paginaverzameling te kennen, omdat de volgende pagina's via de wijzers te bereiken zijn.) Het antwoord is dat op een vaste plaats op schijf (vaak cilinder 0, spoor 0, blok 0) pagina 0 is opgeslagen, die de gewenste informatie bevat. Deze pagina (de schijf-directory of inhoudsopgave genaamd) bevat een lijst van bestaande paginaverzamelingen met een wijzer naar de eerste pagina van die verzameling, zie Fig. 11.6.

Nu richten we onze aandacht op de bestandbeheerder. Zoals de schijfbeheerder het de bestandbeheerder mogelijk maakt om bepaalde apparatuurafhankelijke details te negeren, zo stelt de bestandbeheerder het DBMS in staat om de details met betrekking tot de in- en uitvoer van pagina's te negeren en om te denken in termen van opgeslagen bestanden en records. Deze functie van de bestandbeheerder wordt het *beheer van de opgeslagen records* genoemd. Over die functie zal nu

kort iets worden gezegd; ter illustratie wordt weer de leveranciers-artikelen database genomen.

Veronderstel (en dat is veel realistischer) dat op een pagina vele records opgeslagen kunnen worden in plaats van slechts één, zoals hiervoor werd aangenomen. Veronderstel eveneens dat de gewenste logische volgorde van de leverancierrecords (net als hiervoor) de volgorde op leveranciernummer is. Beschouw nu de volgende gebeurtenissen:

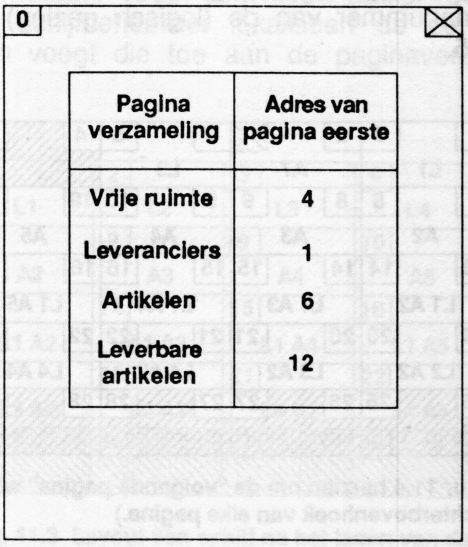



**Fig. 11.6 De schijf - directory (Pagina 0).**

1. De records voor de vijf leveranciers L1-L5 worden ingevoegd en op één pagina, zeg pagina p, geplaatst. Zie Fig. 11.7. Merk op dat pagina p nog steeds een behoorlijke hoeveelheid vrije ruimte bevat (en dus is er per pagina ook een administratie van de vrije ruimte nodig).

2. Veronderstel nu, dat het DBMS een nieuw leverancierrecord toevoegt (voor leverancier L9). De bestandbeheerder slaat dit record op pagina p op (aangezien daar nog ruimte is) en wel onmiddellijk volgend op het opgeslagen record van leverancier L5.

3. Vervolgens verwijdert het DBMS het opgeslagen record van leverancier L2. De bestandbeheerder maakt de ruimte op pagina p, die door het record was bezet, weer vrij en schuift de records van

L3, L4, L5 en L9 op zodat het door record L2 achtergelaten gat wordt opgevuld.

4. Het DBMS voegt opnieuw een record toe, nu voor leverancier L7. De bestandbeheerder plaatst dit record op pagina p direct volgend op het record van leverancier L5. Hierbij wordt het L9 record opgeschoven om ruimte te maken. De situatie is dan zoals door Fig. 11.8 wordt weergegeven.

| P | (Rest van de kop) | | | | | | |
|---|---|---|---|---|---|---|---|
| L1 | Smit | 20 | Londen | L2 | Janse | 10 | Parijs |
| L3 | Blaak | 30 | Parijs | L4 | Klerk | 20 | Londen |
| L5 | Adams | 30 | Athene | | | | |

Fig. 11.7 Layout van pagina P na het laden van de vijf leverancierrecords.

| P | (Rest van de kop) | | | | | | |
|---|---|---|---|---|---|---|---|
| L1 | Smit | 20 | Londen | L3 | Blaak | 30 | Parijs |
| L4 | Klerk | 20 | Londen | L5 | Adams | 30 | Athene |
| L7 | ..... | .... | ..... | L9 | ..... | .... | ..... |

Fig. 11.8 Layout van pagina P na het invoegen van L9, verwijderen van L2 en invoegen van L7.

Dit voorbeeld illustreert dat de logische volgorde van de opgeslagen records *binnen een pagina* kan worden gehandhaafd door de *fysieke* volgorde. De fysieke volgorde op een pagina geeft dus de logische volgorde. De bestandbeheerder kan dit bereiken door, indien nodig, records zodanig op de pagina te schuiven dat ze in de juiste volgorde aan de bovenkant van de pagina komen. De vrije ruimte komt dan aan de onderkant. Merk op dat de logische volgorde van de opgeslagen records over de pagina's heen natuurlijk wordt gerepresenteerd door de volgorde van de pagina's in de betreffende paginaverzameling.

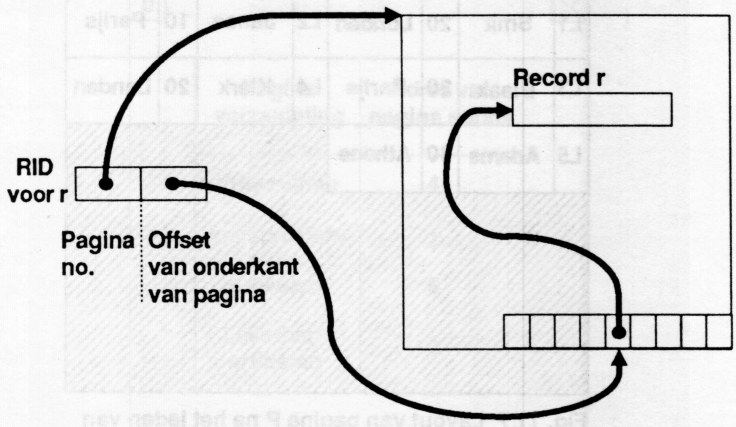



**Fig. 11.9 Implementatie van record - IDs (RIDs).**

Zoals al in paragraaf 11.2 is aangegeven, worden opgeslagen records intern in het database systeem geïdentificeerd door een "record-ID" of RID. Fig. 11.9 toont hoe RID's zouden kunnen worden geïmplementeerd. Het RID van een opgeslagen record r bestaat uit twee delen: het paginanummer p van de pagina die r bevat en een byte offset. Onderaan een pagina bevindt zich een kleine tabel die wijzers naar plekken op de pagina bevat. De byte offset lokaliseert een van die wijzers en die geeft het beginpunt van het record op de pagina. Records kunnen vrijelijk in een pagina worden geschoven zonder dat de RID hoeft te veranderen: alleen de kleine tabel op de pagina moet worden aangepast. Gegeven een RID is toegang tot een opgeslagen record snel, omdat het paginanummer in de RID meteen de goede pagina geeft. Gegeven een RID zal dus in het algemeen (er kunnen zich complicaties voordoen die buiten de scope van dit boek vallen) slechts één paginatoegang nodig zijn om het record op te halen. De RID's worden vaak gebruikt als wijzers naar records, bijvoorbeeld in indexen (zie paragraaf 11.4). Het is dan ook zeer belangrijk dat RID's niet wijzigen, ook niet als het bijbehorende record verschoven moet worden,

omdat anders alle wijzers naar dit verschoven record aangepast zouden moeten worden en dat is een tijdrovende operatie.

Vanzelfsprekend zijn er meerdere RID-implementaties, die alle hun voor en tegen hebben. Het voert echter te ver om daar uitgebreider op in te gaan.

Vanaf nu nemen we aan dat ieder record wordt geïdentificeerd door een RID, die niet wijzigt zolang dat record blijft bestaan. Bovendien nemen we aan, dat een opgeslagen bestand een collectie records is die is opgeslagen in *volgorde van de primaire sleutel*. Deze laatste aanname is alleen gedaan om de hierna volgende uiteenzettingen te vereenvoudigen: er zijn vele andere manieren om records fysiek op te slaan, bijvoorbeeld overeenkomstig het moment van creatie (de zogenaamde *chronologische volgorde*).

In de rest van dit hoofdstuk beschrijven we enige belangrijke toegangstechnieken: indexering, hashing en wijzerketens. Wij wijzen erop dat deze technieken elkaar niet uitsluiten: het is zeer wel mogelijk om een hashed bestand te voorzien van enige indexen en wijzerketens. Het volgende zal een en ander verduidelijken.

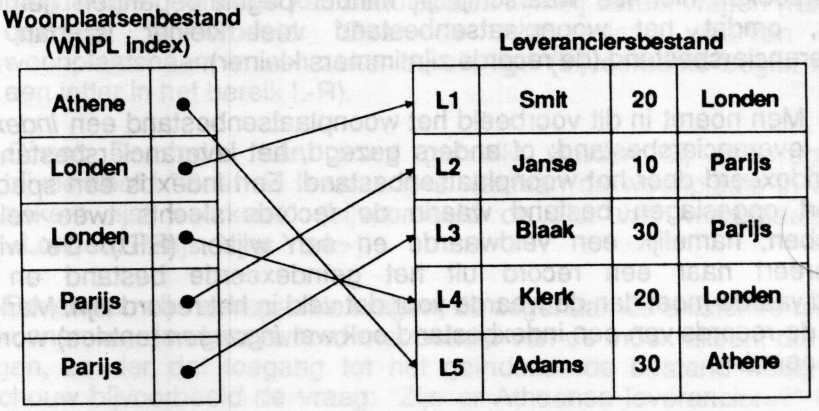



**Fig. 11.10 WNPL index op het leveranciersbestand.**

## 11.4 INDEXERING

Beschouw de leveranciers-artikelen database en veronderstel dat de vraag "Geef alle leveranciers met woonplaats W" heel belangrijk is, bijvoorbeeld omdat hij heel vaak beantwoord moet worden. Het is dan

wenselijk dat deze vraag snel afgehandeld wordt. Om dat te bereiken kan de DBA de representatie van Fig. 11.10 kiezen. In die representatie zijn er twee bestanden, namelijk een leveranciers- en een woonplaatsenbestand (waarschijnlijk in verschillende paginaverzamelingen); het woonplaatsenbestand is opgeslagen in woonplaatsvolgorde (omdat WNPL de primaire sleutel is; zie de aanname zoals die is geformuleerd op het eind van de vorige paragraaf) en bevat wijzers (RID's) naar het leveranciersbestand. Om alle leveranciers in bijvoorbeeld Londen te vinden, heeft het DBMS twee mogelijke strategieën ter beschikking:

1. Doorzoek het hele leveranciersbestand en vind alle records met WNPL = Londen.
2. Zoek in het woonplaatsenbestand naar de records met de waarde Londen en volg voor ieder gevonden record de wijzer naar het corresponderende record in het leveranciersbestand.

Als het aantal Londense leveranciers in verhouding tot de andere klein is, is de tweede strategie waarschijnlijk de snelste, omdat 1) het DBMS de fysieke volgorde van het woonplaatsenbestand kan gebruiken om te stoppen zodra er een record met WNPL > "Londen" is gevonden en 2) zelfs als het DBMS het hele woonplaatsenbestand moet doorzoeken hiermee waarschijnlijk minder paginatoegangen gemoeid zijn, omdat het woonplaatsenbestand veel kleiner is dan het leveranciersbestand (de records zijn immers kleiner).

Men noemt in dit voorbeeld het woonplaatsenbestand een *index* op het leveranciersbestand; of anders gezegd, het leveranciersbestand is geïndexeerd door het woonplaatsenbestand. Een index is een speciaal soort opgeslagen bestand waarin de records slechts twee velden hebben, namelijk een veldwaarde en een wijzer (RID). De wijzer refereert naar een record uit het geïndexeerde bestand en de veldwaarde moet dan de waarde voor dat veld in het record zijn. Merk op dat de records van een indexbestand ook wel *ingangen* (entries) worden genoemd.

Van nu af aan zullen we het indexbestand van Fig. 11.10 aanduiden met de naam "WNPL-index". Een index op een primaire sleutel (zoals bijvoorbeeld LNO in het leveranciersbestand) wordt een *primaire* index genoemd. Een index op een ander veld (bijvoorbeeld de WNPL-index in ons voorbeeld) heet een *secondaire* index.

**11.4.1 Het Gebruik van Indexen.** Het fundamentele voordeel van een index is dat leesoperaties erdoor versneld worden. Er is echter ook een nadeel, namelijk dat bijwerkoperaties worden vertraagd. Ga maar eens na wat het DBMS aan onderhoud van de WNPL-index (van Fig. 11.10) moet doen als een leverancier van Parijs naar Londen verhuist.

Bovendien moet ook bij ieder te verwijderen of in te voegen record de index worden bijgewerkt. Als men overweegt om een bepaald veld van een index te voorzien, dan is het van belang de volgende vraag te beschouwen: wat is belangrijker, snelle leesoperaties gebaseerd op waarden van het beschouwde veld of de vertraging die bij het bijwerken optreedt ten gevolge van de aangebrachte index?

Voor de rest van deze paragraaf concentreren we ons op leesoperaties.

Indexen kunnen op twee manieren worden gebruikt. Ten eerste om *sequentiële toegang* tot het geïndexeerde bestand te bewerkstelligen (waarbij "sequentieel" betekent: in (opklimmende of afdalende) volgorde van het geïndexeerde veld). De WNPL-index in het hiervoor gebruikte voorbeeld staat toe dat de records van het leveranciersbestand in volgorde van woonplaats worden gelezen. Ten tweede kunnen indexen ook worden gebruikt om *directe toegang* tot individuele records van het geïndexeerde bestand te realiseren op basis van een gegeven waarde van het geïndexeerde veld. De vraag "Geef alle leveranciers uit Londen" is daar een voorbeeld van.

Feitelijk kunnen de twee manieren om een index te gebruiken enigszins gegeneraliseerd worden:

1. *Sequentieel:* De index kan behulpzaam zijn bij *bereik-vragen* (range queries); bijvoorbeeld "Geef de leveranciers waarvan de woonplaatsnaam in een zeker bereik ligt" (bijvoorbeeld begint met een letter in het bereik L-R).

2. *Direct:* De index kan zinvol gebruikt worden bij *lijst-vragen*; bijvoorbeeld "Geef de leveranciers waarvan de woonplaats in een zekere lijst voorkomt" (bijvoorbeeld de lijst van woonplaatsen <Londen, Parijs, New York>).

Bovendien zijn er bepaalde vragen, de zogenaamde *existentie-tests*, die kunnen worden beantwoord met behulp van de index alleen, dat wil zeggen, zonder dat toegang tot het geïndexeerde bestand nodig is. Beschouw bijvoorbeeld de vraag: "Zijn er Atheense leveranciers?" Het antwoord op deze vraag is "ja" dan en slechts dan als er een ingang voor Athene in de WNPL-index bestaat.

Een opgeslagen bestand kan vele indexen hebben. Zo kan het leveranciersbestand een WNPL en een STATUS-index hebben, zie Fig. 11.11. Die indexen kunnen dan worden gebruikt om efficiënte toegang tot leverancierrecords te verkrijgen op basis van gegeven waarden voor WNPL en/of STATUS. Beschouw bijvoorbeeld de vraag "Geef alle Parijse leveranciers met een status van 30". De WNPL-index geeft de RID's, namelijk r2 en r3, van de Parijse leveranciers; op dezelfde wijze geeft de STATUS-index de RID's r3 en r5 voor de leveranciers met

status 30. Het is nu duidelijk dat alleen het record met RID r3 aan beide voorwaarden voldoet; het DBMS kan het gewenste record nu direct (laten) ophalen.

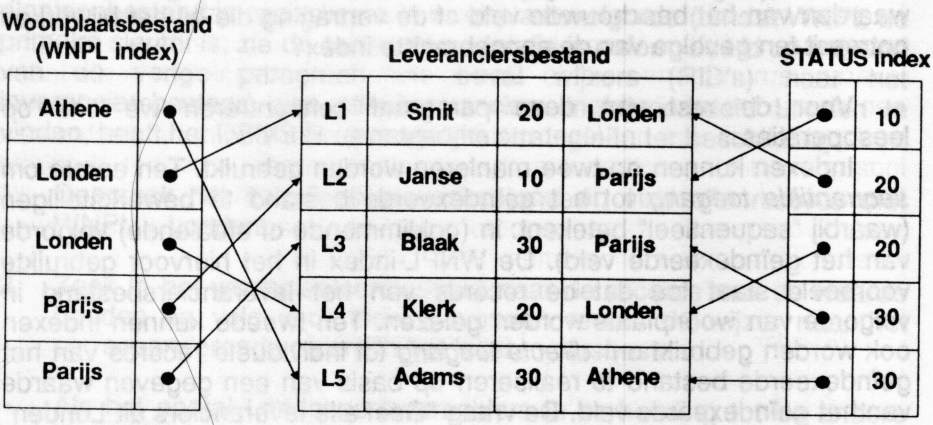



**Fig. 11.11 WNPL en STATUS index op het leveranciersbestand.**

Een index wordt ook wel een *geïnverteerde lijst* genoemd en wel om de volgende reden. Een "normaal" bestand (bijvoorbeeld het leveranciersbestand van Fig. 11.10 en 11.11) geeft voor ieder record de waarden van de velden in dat record. Een index daarentegen bevat voor iedere waarde van het geïndexeerde veld de wijzers naar de records (dat wil zeggen de RID's) die genoemde waarde bezitten. Dit is een vorm van inversie (omkering); vandaar de naam. Als een bestand een index op ieder veld heeft, dan wordt zo'n bestand ook wel *volledig geïnverteerd* genoemd.

**11.4.2 Dichte versus niet-dichte Indexen.** Zoals al een aantal keren is aangegeven, is het doel van een index om het leesproces te versnellen of, meer specifiek, om het aantal paginatoegangen te verkleinen dat nodig is om een record op te halen. In essentie wordt dit doel bereikt door gebruik te maken van *wijzers*. Tot nu toe hebben we aangenomen dat deze wijzers *record-wijzers* zijn (bijvoorbeeld RID's). In feite zijn *pagina-wijzers* (bijvoorbeeld paginanummers) voldoende. Dan zou natuurlijk het DBMS nog extra werk moeten doen om het gewenste record op de gelezen pagina te bepalen. Dat kan echter in het hoofdgeheugen gebeuren, zodat het aantal paginatoegangen niet verandert.

Hierop kunnen we voortborduren. Ieder opgeslagen bestand heeft een "fysieke volgorde" die wordt gerepresenteerd door de combinatie van 1) de volgorde van de opgeslagen records op een pagina en 2) de volgorde van de pagina's in de bevattende paginaverzameling.

Veronderstel nu dat het leveranciersbestand zodanig is opgeslagen dat de fysieke volgorde ervan overeenkomt met de logische volgorde zoals die is gedefinieerd door de waarde van een zeker veld (bijvoorbeeld het veld leveranciernummer). Met andere woorden, het leveranciersbestand is gegroepeerd op dat veld (zie de bespreking van groepering in paragraaf 11.2). Veronderstel ook, dat een index op het leveranciernummer vereist is. Dan is er geen noodzaak om in die index een ingang (record) op te nemen voor ieder record uit het geïndexeerde bestand (dat is dus het leveranciersbestand). Per *pagina* is slechts een ingang nodig, die behalve het paginanummer ook het hoogste leveranciernummer op die pagina bevat. Zie Fig. 11.12, waar ter wille van de eenvoud is aangenomen dat een pagina maximaal twee leverancierrecords kan bevatten.

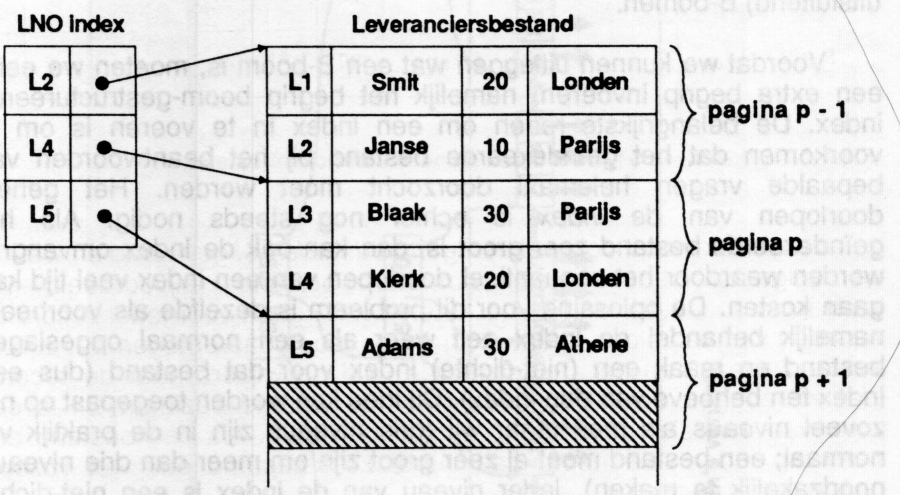



**Fig. 11.12 Voorbeeld van een niet - dichte index.**

Beschouw als voorbeeld wat er moet gebeuren om het record van leverancier L3 met gebruikmaking van deze index te lezen. Eerst moet het systeem de index aflopen om de eerste ingang met een leveranciernummer groter dan of gelijk aan L3 te vinden. De ingang voor leverancier L4 wordt gevonden en die wijst bijvoorbeeld naar pagina p. Het systeem leest pagina p en zoekt in het hoofdgeheugen naar het gewenste record L3.

Een index zoals die van Fig. 11.12 wordt *niet-dicht* genoemd, omdat deze index niet voor ieder record van het geïndexeerde bestand een ingang bevat. Alle tot nu toe besproken indexen heten daarentegen *dichte* indexen. Een voordeel van een niet-dichte index is natuurlijk dat hij minder ruimte inneemt dan een overeenkomstige dichte index (er zijn immers minder ingangen). Een gevolg daarvan is dat het aflopen van

een niet-dichte index ook sneller gaat. Een nadeel is dat het niet langer mogelijk is om existentie-tests uit te voeren op basis van de index alleen.

Merk op dat een opgeslagen bestand ten hoogste één niet-dichte index kan hebben, omdat zo'n index gebaseerd is op de unieke fysieke volgorde van het bestand in kwestie. Alle andere indexen moeten dus dicht zijn.

**11.4.3 B-bomen.** Indexen zijn er in vele soorten en een belangrijke en veel gebruikte soort is de B-boom. Er is reeds eerder opgemerkt dat er geen enkele opslagstructuur is die optimaal is voor alle toepassingen. Toch bestaat er weinig twijfel over dat als één opslagstructuur gekozen zou mogen worden, dit de B-boom (in de een of andere vorm) zou zijn. B-bomen lijken de beste 'all-round' prestatie te leveren. Om deze reden ondersteunen de meeste relationele systemen (en sommige zelfs uitsluitend) B-bomen.

Voordat we kunnen uitleggen wat een B-boom is, moeten we eerst een extra begrip invoeren, namelijk het begrip boom-gestructureerde index. De belangrijkste reden om een index in te voeren is om te voorkomen dat het geïndexeerde bestand bij het beantwoorden van bepaalde vragen helemaal doorzocht moet worden. Het geheel doorlopen van de *index* is echter nog steeds nodig. Als het geïndexeerde bestand zeer groot is, dan kan óók de index omvangrijk worden waardoor het sequentieel doorlopen van een index veel tijd kan gaan kosten. De oplossing voor dit probleem is dezelfde als voorheen, namelijk behandel de index zelf weer als een normaal opgeslagen bestand en maak een (niet-dichte) index voor dat bestand (dus een index ten behoeve van een index). Dit idee kan worden toegepast op net zoveel niveaus als men maar wil (drie niveaus zijn in de praktijk vrij normaal; een bestand moet al zeer groot zijn om meer dan drie niveaus noodzakelijk te maken). Ieder niveau van de index is een niet-dichte index voor het niveau eronder. Op deze wijze ontstaat een zogenaamde boom-gestructureerde index.

Nu kunnen we overgaan naar de B-bomen. Een B-boom is een speciaal soort boom-gestructureerde index. B-bomen komen in allerlei varianten voor. In deze paragraaf zullen we de variant voorgesteld door D.E. Knuth beschrijven.

In Knuth's variant bestaat de index uit twee delen, namelijk de *sequence set* en de *index set*.

1. *De sequence set* bestaat uit een index met slechts één niveau van waaruit naar de records van het bestand wordt verwezen. Deze index is normaal gesproken dicht, maar zou niet-dicht kunnen zijn als het geïndexeerde bestand gegroepeerd is op het geïndexeerde veld. De pagina's van de index zijn (natuurlijk) geketend.

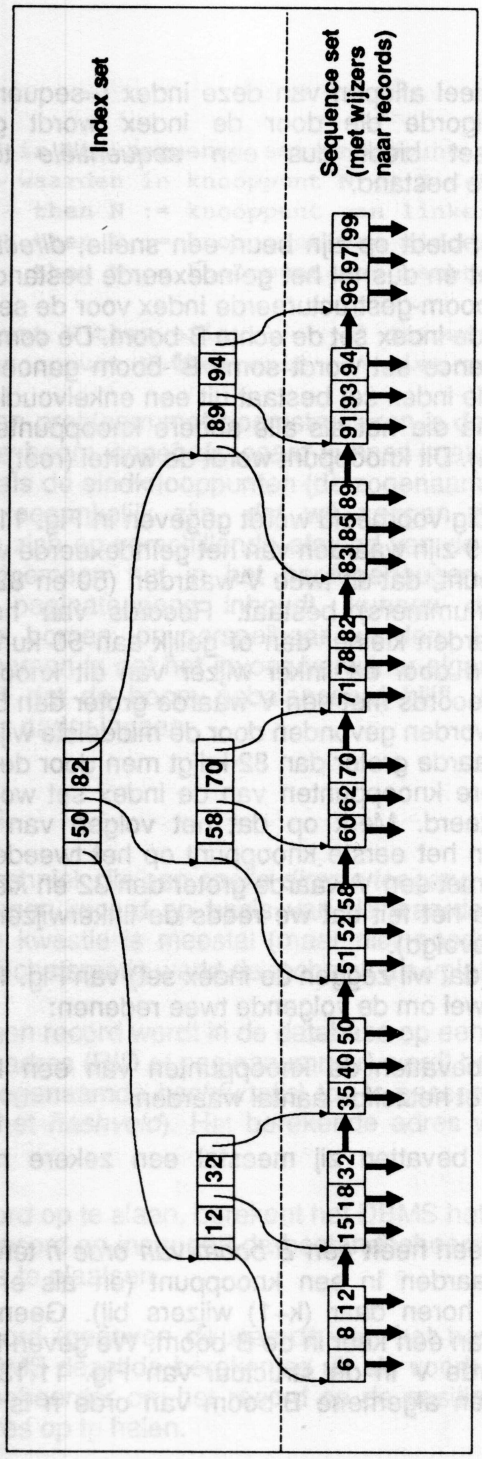



Fig. 11.13 Deel van een B - boom (Knuth's variant).

Het sequentieel aflopen van deze index (=sequence set) geeft de logische volgorde die door de index wordt gedefinieerd. De sequence set biedt dus een *sequentiële* toegang tot het geïndexeerde bestand.

2. De *index set* biedt op zijn beurt een snelle, *directe* toegang tot de sequence set en dus tot het geïndexeerde bestand. De index set is in feite een boom-gestructureerde index voor de sequence set; strikt genomen is de index set de echte B-boom. De combinatie van index set en sequence set wordt soms $B^*$-boom genoemd. Het hoogste niveau van de index set bestaat uit een enkelvoudig knooppunt (dat is een pagina die net als alle andere knooppunten vele ingangen kan bevatten). Dit knooppunt wordt de wortel (root) genoemd.

Een eenvoudig voorbeeld wordt gegeven in Fig. 11.13. De waarden 6, 8, 12, ..., 97, 99 zijn waarden van het geïndexeerde veld V. Beschouw het wortelknooppunt, dat uit twee V-waarden (50 en 82) en drie wijzers (dit zijn paginanummers) bestaat. Records van het geïndexeerde bestand met waarden kleiner dan of gelijk aan 50 kunnen (uiteindelijk) worden gevonden door de linker wijzer van dit knooppunt te volgen. Evenzo kunnen records met een V-waarde groter dan 50 en kleiner dan of gelijk aan 82 worden gevonden door de middelste wijzer te volgen. De records met V-waarde groter dan 82 krijgt men door de rechter wijzer te volgen. De andere knooppunten van de index set worden op analoge wijze geïnterpreteerd. Merk op dat het volgen van bijvoorbeeld de rechter wijzer van het eerste knooppunt op het tweede niveau ons alle records oplevert met een V-waarde groter dan 32 en *kleiner dan of gelijk aan 50* (vanwege het feit dat we reeds de linkerwijzer van het hogere niveau hebben gevolgd).

De B-boom (dat wil zeggen de index set) van Fig. 11.13 is enigszins onrealistisch, en wel om de volgende twee redenen:

1. Ten eerste bevatten de knooppunten van een B-boom normaal gesproken niet hetzelfde aantal waarden.

2. Ten tweede bevatten zij meestal een zekere hoeveelheid vrije ruimte.

In het algemeen heeft een *B-boom van orde n* tenminste n en niet meer dan 2n waarden in een knooppunt (en als er k waarden zijn opgeslagen dan horen daar (k+1) wijzers bij). Geen enkele waarde verschijnt meer dan één keer in de B-boom. We geven het algoritme om een zekere waarde V in de structuur van Fig. 11.13 te zoeken: het algoritme voor een algemene B-boom van orde n is een eenvoudige generalisatie.

```
N := wortel ;
repeat until N is een sequence set knooppunt ;
  X, Y zijn de waarden in knooppunt N (* X < Y *) ;
  if V <= X      then N := knooppunt van linkerwijzer ;
  if X < V <= Y then N := knooppunt van middelste wijzer ;
  if V > Y       then N := knooppunt van rechterwijzer ;
end repeat ;
if V in knooppunt N then exit          (* gevonden *) ;
if V niet in knooppunt N then exit  (* niet gevonden *) ;
```

Een algemeen probleem met boomstructuren is dat invoegingen en verwijderingen de boom *ongebalanceerd* kunnen maken. Een boom is ongebalanceerd als de eindknooppunten (de zogenaamde bladeren) niet meer gelijkelijk toegankelijk zijn, dat wil zeggen dat verschillende eindknooppunten zich op verschillende afstand van de wortel bevinden. Omdat in het algemeen het in het hoofdgeheugen halen van een knooppunt een paginatoegang inhoudt, kunnen de zoektijden in ongebalanceerde bomen onvoorspelbaar worden. Het opmerkelijke voordeel van B-bomen is dat het invoeg/verwijder algoritme van een B-boom garandeert dat de boom gebalanceerd blijft. Op dit algoritme zullen we hier niet nader ingaan.

## 11.5 HASHING

Hashing is een techniek die een snelle *directe* toegang verschaft tot een specifiek opgeslagen record op basis van de waarde voor een zeker veld. Het veld in kwestie is meestal (maar niet noodzakelijkerwijs) de primaire sleutel. Schetsmatig werkt de techniek als volgt.

- Elk opgeslagen record wordt in de database op een positie geplaatst waarvan het adres (RID of paginanummer) wordt berekend door een functie (de zogenaamde *hashfunctie*) toe te passen op een veld van dat record (het *hashveld*). Het berekende adres wordt *hash-adres* genoemd.
- Om een record op te slaan, berekent het DBMS het hash-adres voor het nieuwe record en instrueert de bestandbeheerder om het record op die positie te plaatsen.
- Om een record (gegeven de waarde van het hashveld) te lezen, voert het DBMS dezelfde berekening uit als voorheen en instrueert de bestandbeheerder om het record op de positie aangeduid door het hash-adres op te halen.

Ter illustratie, veronderstel dat 1) de leveranciernummers L100, L200, L300, L400, L500 zijn (in plaats van L1, L2, L3, L4, L5) en dat 2) elk opgeslagen record een gehele pagina in beslag neemt. Beschouw nu de hashfunctie:

hash-adres (d.w.z. paginanummer) :=
rest na deling van het numerieke deel van LNO door 13.

Dit is een eenvoudig voorbeeld van een zeer algemene klasse van hashfuncties, de "division/remainder" hashfuncties genaamd. Nemen we L100, dan levert het delen van 100 door 13 een rest van 9 op. De paginanummers worden respectievelijk 9, 5, 1, 10, 6 (zie Fig. 11.14).

Het moet op grond van het voorgaande duidelijk zijn dat hashing verschilt van indexering in zoverre dat een opgeslagen bestand vele indexen kan hebben, maar daarentegen *ten hoogste* één hashstructuur. Met andere woorden: een bestand kan vele geïndexeerde velden hebben, doch slechts één hashveld.

Dit voorbeeld toont ook aan waarom de hashfunctie nodig is. Het is theoretisch mogelijk om de 'identiteit' hashfunctie te gebruiken, dat wil zeggen om het (numerieke) deel van de waarde van de primaire sleutel van een record meteen als hash-adres te laten fungeren. Een dergelijke techniek zal echter in de praktijk meestal niet goed werken, omdat het aantal elementen in het domein van de primaire sleutel (dat is de verzameling van alle mogelijke sleutelwaarden) normaal gesproken veel en veel groter is dan het aantal beschikbare adressen. In termen van het voorbeeld: de leveranciernummers bevatten drie cijfers, dus zijn er 1000 verschillende leveranciernummers, terwijl er in feite maar een tiental leveranciers is. Om een behoorlijke verspilling van opslagcapaciteit te vermijden, zouden we graag een hashfunctie hebben die een waarde in het bereik van 000-999 terugbrengt tot een waarde in het bereik van bijvoorbeeld 0-9. Om een beetje toekomstige groei toe te staan (denk aan invoegingen), is het gebruikelijk om een 20 procent extra ruimte te nemen. Daarom gebruikten we in het voorbeeld hierboven een functie die waarden in het bereik 0-12 genereerde (en niet in het bereik 0-9).

Let wel: het is realistischer te veronderstellen dat een primaire sleutel bijvoorbeeld 8 cijfers bevat en dat het aantal records dat opgeslagen moet worden in de orde van enige (tien)duizenden ligt. Echter, de discrepantie tussen het aantal mogelijke sleutelwaarden en het actuele aantal wordt hierdoor niet kleiner.

Fig. 11.14 Voorbeeld van een hashstructuur.

Het voorbeeld toont ook een van de nadelen van hashing: de "fysieke volgorde" van de records in een opgeslagen bestand zal vrijwel zeker niet de primaire sleutel volgorde zijn, maar ook niet een andere volgorde die een zinnige logische interpretatie heeft. Bovendien kunnen er "gaten" tussen opeenvolgende records bestaan (zie Fig. 11.14). In feite wordt er gewoonlijk vanuit gegaan dat de fysieke volgorde in een bestand met een hashed structuur niet een bepaalde logische volgorde heeft. Zo zal een hashed bestand normaal gesproken geen intra-bestand groepering hebben, hetgeen nogal ongelukkig is omdat fysieke groepering, zoals al eerder opgemerkt, in het algemeen zeer gewenst is.

Een ander nadeel van hashing is dat er altijd het gevaar van *collisie* (botsing, in het Engels collision) is, dat wil zeggen dat twee verschillende records na hashing op hetzelfde adres terecht komen. Veronderstel bijvoorbeeld dat het leveranciersbestand (met leveranciers L100, L200, etc.) ook een leverancier met nummer L1400 bevat. Deze leverancier zou, gegeven de "deel door 13" hashfunctie, op hetzelfde adres als leverancier L100 worden afgebeeld, namelijk hash-adres 9. De tot nu toe beschreven hashing techniek moet dus worden uitgebreid om het probleem van collisie aan te kunnen. Een mogelijke uitbreiding is om de rest na deling door 13 niet te beschouwen als het hash-adres zonder meer, maar als het startpunt voor een sequentiële zoekactie. Om bijvoorbeeld het record van leverancier L1400 (aangenomen dat de leveranciers L100-L500 al bestaan) in te voegen, gaan we naar pagina 9, vinden deze reeds bezet en zoeken dan vanaf deze positie in voorwaartse richting naar een vrije pagina. De eerste vrije pagina is pagina 11 en daar wordt het record van L1400 dan opgeslagen. Als we een record gaan lezen, dan hashen we de waarde van het hashveld, gaan naar de gevonden pagina en als het daar opgeslagen record niet het gezochte is, dan zoeken we vanaf deze pagina verder totdat we het gezochte record gevonden hebben.

Enige terminologie: genoemd zoekproces staat (om voor de hand liggende redenen) bekend onder de naam *lineair zoeken*. Het record van leverancier L1400 noemt men een *overlooprecord* (overflow record) en de techniek om een overlooprecord op de eerste vrije plaats vanaf het hash-adres op te slaan, noemt men de *progressieve* overlooptechniek.

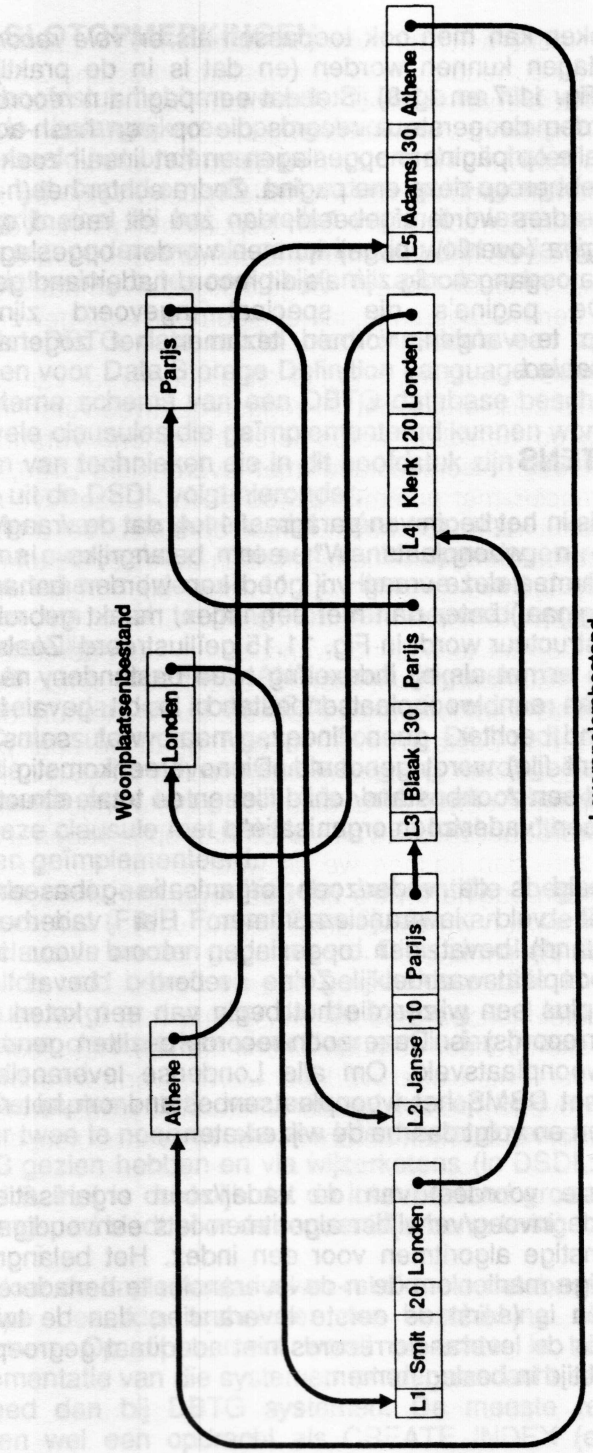



Fig. 11.15 Voorbeeld van een vader/zoon structuur.

Het lineair zoeken kan men ook toepassen als er *vele records op één pagina* opgeslagen kunnen worden (en dat is in de praktijk erg waarschijnlijk, zie Fig. 11.7 en 11.8). Stel dat een pagina n records kan bevatten. Dan worden de eerste n records die op een hash-adres p worden afgebeeld alle op pagina p opgeslagen en het lineair zoeken zal alleen betrekking hebben op deze ene pagina. Zodra echter het (n+1)-de record op dit hash-adres wordt afgebeeld, dan zou dit record op een aparte *overlooppagina* (overflow page) kunnen worden opgeslagen en zal een extra paginatoegang nodig zijn als dit record naderhand gelezen moet worden. De pagina's die speciaal ingevoerd zijn om overlooprecords op te vangen, vormen tezamen het zogenaamde *separate overloopgebied*.

## 11.6 WIJZERKETENS

Veronderstel, net als in het begin van paragraaf 11.4, dat de vraag "Geef alle leveranciers in woonplaats W" een belangrijke is. Een opslagstructuur waarmee deze vraag vrij goed kan worden behandeld, mogelijk zelfs (marginaal) beter dan met een index, maakt gebruik van wijzerketens. Zo'n structuur wordt in Fig. 11.15 geïllustreerd. Zoals daar is aangegeven, zijn er net als bij indexering twee bestanden, namelijk een leveranciers- en een woonplaatsenbestand. In dit geval is het woonplaatsenbestand echter geen index, maar wat soms een vaderbestand (parent file) wordt genoemd. Dienovereenkomstig is het leveranciersbestand een zoonbestand (child file) en de totale structuur is een voorbeeld van een "vader/zoon organisatie".

In het voorbeeld is de vader/zoon organisatie gebaseerd op waarden van het veld leveranciernummer. Het vaderbestand (=woonplaatsenbestand) bevat een opgeslagen record voor iedere verschillende woonplaatswaarde. Zo'n record bevat een woonplaatswaarde plus een wijzer die het begin van een keten zoon-records (leverancierrecords) is. Deze zoon-records bezitten genoemde waarde voor het woonplaatsveld. Om alle Londense leveranciers te vinden, doorzoekt het DBMS het woonplaatsenbestand om het record van Londen te vinden en volgt daarna de wijzerketen.

Het voornaamste voordeel van de vader/zoon organisatie (de wijzerketen) is dat de invoeg/verwijder algoritmen iets eenvoudiger zijn dan de overeenkomstige algoritmen voor een index. Het belangrijkste nadeel is dat de enige manier om de n-de leverancier te benaderen het volgen van de keten is (eerst de eerste leverancier, dan de tweede leverancier, etc.). Als de leverancierrecords niet adequaat gegroepeerd zijn, dan kan dit veel tijd in beslag nemen.

## 11.7 SLOTOPMERKINGEN

In dit hoofdstuk hebben we een bij lange na niet volledige blik op enige van de belangrijkere opslagstructuren geworpen. Zo hebben we bijvoorbeeld het onderwerp geheugengebruik helemaal niet beschouwd, terwijl het beperken van het extern geheugengebruik vaak toch een belangrijk doel bij het opstellen van het interne schema is. Daardoor kwam een onderwerp als comprimering (het zodanig opslaan van gegevens dat weinig bytes nodig zijn) niet aan de orde.

Het DBTG data model biedt inmiddels een zogenaamde DSDL, hetgeen voor Data Storage Definition Language staat. In deze taal wordt het interne schema van een DBTG database beschreven. Deze DSDL kent vele clausules die geïmplementeerd kunnen worden door gebruik te maken van technieken die in dit hoofdstuk zijn beschreven. Een (kleine) greep uit de DSDL volgt hieronder.

Men kan aangeven, dat één recordtype van het conceptuele schema overeenkomt met een aantal opgeslagen recordtypen van het interne schema (zie ook 11.1). Hiermee heeft de DBA een middel in handen om snel toegang tot een bepaald opgeslagen bestand te bewerkstelligen.

Als vaak een FIND ANY wordt gegeven met een zeker veld als parameter, dan kan in de DSDL voor dit veld een LOCATION MODE IS CALC clausule worden gespecificeerd. Dit houdt in dat, gegeven een waarde voor dit veld, een snelle toegang wordt geëist tot een record dat die waarde voor het genoemde veld heeft. Afhankelijk van het DBMS zou deze clausule met behulp van een hashfunctie of een index kunnen worden geïmplementeerd.

Als vaak een conditionele, sequentiële toegang binnen een set wordt vereist (FIND Format 2 en 6), dan kan in de DSDL een SEARCH KEY clausule worden gedefinieerd. Dit is zinvol als de conditie bijna altijd hetzelfde veld betreft en een gelijkheidsconditie is. De SEARCH KEY clausule zorgt er dan voor dat de toegang snel kan worden uitgevoerd. Ook deze clausule kan weer door een index (maar nu per set!) worden gerealiseerd.

De implementatie van een set kan op vele manieren geschieden. Om er twee te noemen: via een inter-bestand groepering, zoals we dat in 11.2.3 gezien hebben en via wijzerketens (in DSDL: MODE IS CHAIN), zie 11.6. Het is duidelijk dat de inter-bestand groepering een zeer snel toegangspad biedt voor de sequentiële benadering van de set.

Voor de implementatie van relationele database systemen staan in principe dezelfde technieken ter beschikking als voor de DBTG systemen. Op dit moment wordt er echter in het algemeen bij de implementatie van die systemen minder aandacht aan prestatieaspecten besteed dan bij DBTG systemen. De meeste relationele systemen kennen wel een opdracht als CREATE INDEX (en zijn tegenhanger

DROP INDEX). Meestal zal, zoals al in dit hoofdstuk is opgemerkt, dit neerkomen op de creatie van een B-boom voor het betreffende attribuut. In sommige relationele systemen worden hashing en wijzerketens aangeboden.

We besluiten dit hoofdstuk met het benadrukken van het feit dat de meeste gebruikers niets behoeven te weten van de stof zoals die in dit hoofdstuk is gepresenteerd. De enige "gebruiker" die de stof wel moet begrijpen is de DBA, die verantwoordelijk is voor het fysieke ontwerp van de database en voor het meten en verbeteren van de prestatie.

## OPGAVEN

**11.1** Beschouw de vraag "Geef alle Londense leveranciers". Stel dat er een index op WNPL aanwezig is. Gevraagd wordt nu hoe het DBMS de RID's van de records van de Londense leveranciers bepaalt. De tweede vraag is, hoe de bestandbeheerder de pagina bepaalt waarop het record met een zekere RID is opgeslagen.

**11.2** Op een schijvengeheugen is een bestand B opgeslagen. Het bestand vult volledig de cilinders 20 tot en met 39 en 50 tot en met 89. Op het schijvengeheugen wordt een groot aantal toegangen uitgevoerd, die uitsluitend betrekking hebben op records van B. Gegeven is dat de kans op toegang voor alle records van B gelijk is en dat alle cilinders evenveel records bevatten. Na elke toegang blijft de arm met de lees/schrijfkoppen staan op de cilinder waarop de laatste toegang heeft plaatsgevonden.
Bereken het gemiddeld aantal cilinders dat per toegang overbrugd moet worden.

**11.3** Een opgeslagen bestand van records zonder additionele toegangsstructuur (dat wil zeggen geen index, geen hashing, etc.) wordt soms een *heap* genoemd. Nieuwe records worden in de heap gevoegd op die plaats waar toevallig ruimte is. Voor bestanden met een klein aantal (zeg twee, drie) pagina's is de heap waarschijnlijk een zeer efficiënte structuur. De meeste bestanden zijn echter groter. Vergelijk de voor- en nadelen van een heap structuur met een bestand waarbij een index op de primaire sleutel aanwezig is.

**11.4** In paragraaf 11.5 werd de progressieve overlooptechniek beschreven. Kunt u aangeven hoe verwijderoperaties moeten worden uitgevoerd. Als er meer varianten zijn, geef dan per variant aan hoe de bijbehorende leesoperatie eruit ziet.

**11.5** Gegeven is een bestand waarvan de records uniek worden geïdentificeerd door een sleutel.

In welke van de drie gevallen heeft het zin om dit bestand op sleutel te sorteren.
a. Het zoeken van individuele records (waarbij de sleutelwaarde gegeven is) vindt af en toe plaats, terwijl het niet zeker is of het gezochte record in het bestand voorkomt.
b. Idem, maar nu weten we steeds zeker dat het gezochte record voorkomt.
c. Het zoeken van individuele records wordt uitgesteld tot er een groep sleutelwaarden voorhanden is. Als deze groep groot genoeg is, worden de zoekoperaties uitgevoerd.
Beargumenteer uw antwoord.

**11.6** Beschouw de niet-dichte index zoals beschreven in 11.4.2. Een dergelijke index bevat per pagina van het geïndexeerde bestand een ingang, bestaande uit een veldwaarde en een wijzer, en maakt gebruik van het feit dat het bestand gesorteerd is op het geïndexeerde veld.
Stel nu dat gegeven is de omvang van bestanden niet in hele pagina's maar in hele cilinders wordt uitgedrukt en dat extra ruimte in een of meer cilinders wordt uitgereikt. Gevraagd wordt om voor deze situatie een "verbeterde" niet-dichte index te ontwerpen.

**11.7** Gegeven is een AUTEUR bestand met attributen (ANAAM, ADRES, GEB-DATUM). Hierbij is ANAAM de primaire sleutel. Stel dat het aantal toevoegingen en verwijderingen in het AUTEUR bestand zeer groot is. Neem bovendien aan dat zeer snelle toegang tot een AUTEUR record bij gegeven ANAAM gewenst is. Welke opslagstructuur zou u voor AUTEUR kiezen. Beargumenteer uw antwoord.

# Antwoorden op Geselecteerde Opgaven

## HOOFDSTUK 1

**1.2** Het associatietype HUWELIJK dat mannen en vrouwen associeert.

## HOOFDSTUK 3

**3.1** Relationele visie.

PERSOON

| PNAAM | ADRES | ... |
|---|---|---|
| Arthur | --- | --- |
| Wil | --- | --- |
| Charlie | --- | --- |
| David | --- | --- |

BEKWAAMHEID

| BNAAM | CURSUS | CODE | ... |
|---|---|---|---|
| Programmeren | --- | --- | --- |
| Ontwerpen | --- | --- | --- |
| Analyse | --- | --- | --- |

PERS-BEKW

| PNAAM | BNAAM | DATUM |
|---|---|---|
| Arthur | Programmeren | --- |
| Wil | Ontwerpen | --- |
| Wil | Programmeren | --- |
| Charlie | Analyse | --- |
| Charlie | Programmeren | --- |
| Charlie | Ontwerpen | --- |
| David | Ontwerpen | --- |
| David | Analyse | --- |

**3.2** Twee hiërarchische visies.
De twee mogelijke hiërarchische visies bestaan uit (a) vier bomen van diepte twee, één voor elke persoon, met de bekwaamheden van een persoon ondergeschikt aan die persoon en (b) drie bomen (ook van diepte twee), waarbij de personen ondergeschikt aan de bekwaamheden zijn.

**3.3** Netwerkvisie.
De netwerkvisie bestaat uit vier persoonrecords, drie bekwaamheidrecords en acht verbindingrecords. Elk verbindingrecord 'koppelt' een persoonrecord aan een bekwaamheidrecord en bevat de datum waarop de persooon die bekwaamheid verkreeg.

**3.4** De beantwoording van deze vraag lijkt sterk op de antwoorden op de vragen V1 (of V2) die in dit hoofdstuk zijn behandeld.

Neem als relationele visie de beschrijving als onder 3.1 vermeld. De 'tuple-at-a-time' programmering is analoog aan de uitwerking in Fig. 3.2. De algebraïsche procedure wordt:

```
SELECT   PERS-BEKW  WHERE BNAAM='Analyse' GIVING TEMP
PROJECT  TEMP  OVER  PNAAM GIVING RESULT
```

Als in de hiërarchische database de personen 'boven' de bekwaamheden staan, dan is de uitwerking analoog aan die van vraag V2 in Fig. 3.4, anders aan die van vraag V1 van diezelfde figuur.

Voor de netwerk benadering, zie Fig. 3.6.

## HOOFDSTUK 4

**4.2** Het relationele schema ziet er als volgt uit:

```
DOMAIN     PNAAM      CHARACTER (15)
DOMAIN     ADRES      CHARACTER (40)
DOMAIN     BNAAM      CHARACTER (15)
DOMAIN     CURSUS     CHARACTER (30)
DOMAIN     CODE       CHARACTER (2)
DOMAIN     DATUM      CHARACTER (6)

RELATION - PERSOON   (PNAAM,ADRES,...) ,
         PRIMARY KEY PNAAM

RELATION   BEKWAAMHEID   (BNAAM,CURSUS,CODE,...) ,
         PRIMARY KEY BNAAM
```

```
RELATION  PERS-BEKW  (PNAAM,BNAAM,DATUM) ,
          PRIMARY KEY (PNAAM,BNAAM) ,
          FOREIGN KEY PNAAM IDENTIFIES PERSOON ,
          FOREIGN KEY BNAAM IDENTIFIES BEKWAAMHEID
```

We nemen aan dat elk attribuut is gedefinieerd op het domein met dezelfde naam.

**4.5** Genoemde situatie kan zich wel voordoen; relatie R moet dan wel een tupel met a1 als sleutelwaarde bevatten. Bovendien moet de waarde c1 van de vreemde sleutel C voorkomen in de relatie waar C primaire sleutel is.

**4.6** Relatie A heeft TYPE als primaire sleutel en relatie I de combinatie (TYPE,KLEUR).

a. De relatie A kan maximaal 3 tupels bevatten
b. A kan maximaal 3x4=12 tupels bevatten.
c. Neen.
d. Ja.
e. Ja
f. Neen.

**4.7** Het schema (opnieuw zonder definitie van de domeinen) ziet er als volgt uit.

```
RELATION  L
          ( LNO, LNAAM, STATUS, WNPL ) ,
          PRIMARY KEY LNO

RELATION  A
          ( ANO, ANAAM, KLEUR, GEW, PLAATS ) ,
          PRIMARY KEY ANO

RELATION  P
          ( PNO, PNAAM, STAD ) ,
          PRIMARY KEY PNO

RELATION  LAP
          ( LNO, ANO, PNO, HOEV ) ,
          PRIMARY KEY (LNO,ANO,PNO) ,
          FOREIGN KEY LNO IDENTIFIES L ,
          FOREIGN KEY ANO IDENTIFIES A ,
          FOREIGN KEY PNO IDENTIFIES P
```

## HOOFDSTUK 5

**5.1** De relatie INHOUDSOPGAVE kent als primaire sleutel (HFDST-NO,BOEKNO) en als vreemde sleutel BOEKNO. De INHOUDSOPGAVE tupels die bij een zeker boek behoren zijn zogenaamd bestaansafhankelijk van dat boek, dat wil zeggen, zodra het boek verdwijnt zullen ook de bijbehorende hoofdstukbeschrijvingen verdwijnen. De specificatie wordt:

```
FOREIGN KEY
        BOEKNO  IDENTIFIES  BOEKEN,
        DELETE  OF  BOEKEN CASCADES,
        ON  INSERT  NULL NOT ALLOWED,
        MODIFY ALLOWED ONLY FROM NULL
```

**5.2**

a. De entiteittypen zijn Auteurs, Tijdschriften, Artikelen en Onderwerpen. De hierbij behorende relaties zijn respectievelijk AUTEURS, TIJDSCHRIFTEN, ARTIKELEN en ONDERWERPEN. Tussen Artikelen en Tijdschriften bestaat een m:n associatietype, die we representeren door een relatie die we ART-TIJDSCH noemen. Een m:n associatietype bestaat ook tussen Auteurs en Artikelen. De bijbehorende relatie heet AUT-ART. Bij een artikel hoort slechts één onderwerp, dus het associatietype tussen Onderwerp en Artikel is 1:n. Dit kan worden opgelost door een vreemde sleutel naar het onderwerp in het tupel behorend bij een artikel op te nemen. Rest nog één probleem, namelijk het niet-atomaire kenmerk kernwoorden. Hiervoor creëren we een extra relatie, KERNWOORDEN genaamd. Deze bevat natuurlijk een vreemde sleutel naar het artikel dat beschreven wordt.

b. We krijgen nu het volgende (de primaire sleutels zijn onderstreept, vreemde sleutels gecursiveerd):

| | |
|---|---|
| ONDERWERPEN | (ONDERW-NO, ONAAM) |
| ARTIKELEN | (TITEL, ATYPE, *ONDERW-NO*) |
| AUTEURS | (ANAAM, ADRES, INSTITUUT) |
| TIJDSCHRIFTEN | (TNAAM, TTYPE, UITGEVER) |
| ART-TIJDSCH | (*TITEL, TNAAM*) |
| AUT-ART | (*ANAAM, TITEL*) |
| KERNWOORDEN | (*TITEL,* KERNWOORD) |

**5.3** De relaties behorende bij de entiteittypen worden respectievelijk KLANTEN, ORDERS, en ARTIKELEN genoemd. Een klant kan vele orders plaatsen, echter een order hoort bij slechts een klant (1:n associatietype). Een order heeft een niet-atomair kenmerk, namelijk orderregel. Hiervoor wordt een aparte relatie ORDERREGELS gedefinieerd. Een tupel uit de ORDERREGELS relatie verwijst naar het

bestelde artikel. We krijgen het volgende (ook hier betekent cursief weer 'vreemde sleutel', terwijl onderstreping de primaire sleutel aangeeft):

```
KLANTEN        (KNAAM, ADRES)
ORDERS         (ORDERNO, KNAAM, LEVERDATUM, KORTING)
ARTIKELEN      (ANAAM, GEWICHT)
ORDERREGELS    (ORDERNO, REGELNO, ANAAM, HOEVEELHEID)
```

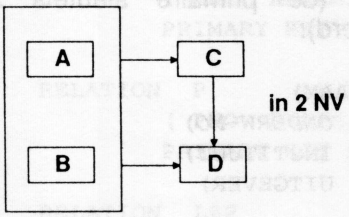
**5.4** Het interessante aspect hier is dat docenten en studenten een aantal kenmerken gemeen hebben, namelijk naam, adres en woonplaats. Men zegt wel dat studenten en docenten specialisaties van algemenere objecten zijn. Deze algemenere objecten zijn generalisaties van studenten en docenten en zullen hier personen worden genoemd. De gemeenschappelijke kenmerken worden aan personen toegekend, de onderscheidende kenmerken aan docenten en studenten. Verder is er een 1:n associatietype tussen Docenten en Colleges en een n:m associatietype tussen Studenten en Colleges. Dus:

```
PERSONEN       (PNAAM, ADRES, WOONPLAATS)
DOCENTEN       (DNAAM, KAMERNUMMER)
STUDENTEN      (SNAAM, LEEFTIJD)
COLLEGES       (CNO, DNAAM, ZAAL, TIJDSTIP)
STUD-COLL      (CNO, SNAAM)
```

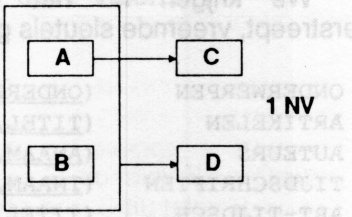
## HOOFDSTUK 6

**6.1**

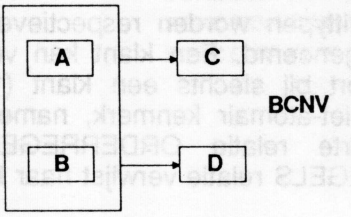
a.



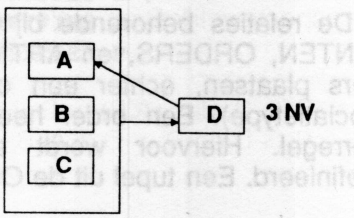

b.



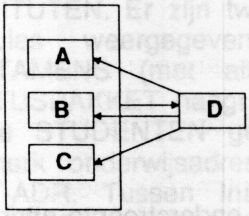

c.




d.



e.




BCNV

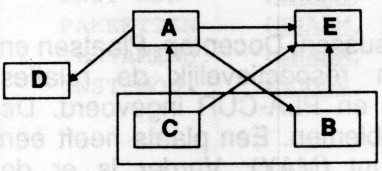
2 Kandidaatsleutels (ABC) en D.

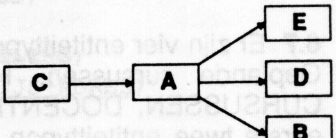
**6.2**

a. Ja.
b. Neen, zie relatie TWEEDE van dit hoofdstuk.
c. Ja.
d. Ja, zie Stelling van Heath, genoemd in dit hoofdstuk.
e. Neen, zie relatie TWEEDE van dit hoofdstuk.
f. Ja (dit is alleen een manier van noteren).
g. Ja (evenzo).
h. Ja.

**6.3** Deze relatie is in BCNV, omdat de twee determinanten A en B in R kandidaatsleutels zijn.

**6.4**

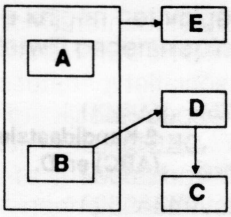



(B,C) -> E volgt uit C -> E en
C -> E volgt uit C -> A en A -> E.
Het is nu eenvoudig in te zien dat C de primaire sleutel van R is. Relatie R is niet in 3NV (transitieve afhankelijkheid), maar wel in 2NV. Door R te splitsen in R1 (C,A) en R2 (A,B,D,E) krijgen we twee relaties in BCNV met respectievelijk C en A als primaire sleutels.

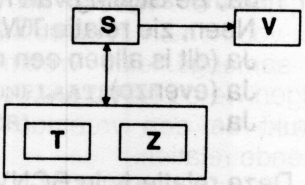
**6.5**

a. B -> D, dus (A,B) -> D
B -> D en D -> C dus B -> C en (A,B) -> C.




b. R1 (A,B,E), R2 (B,D) en R3 (D,C). De onderstreepte attributen vormen de primaire sleutel.
c. Door de extra FA D -> B wordt D in R2 ook kandidaatsleutel. Er hoeft echter geen relatie gecreëerd te worden.

**6.6** De afhankelijkheden zijn de volgende:
S -> V,
(T,Z) -> S
S -> (T,Z)




We krijgen de relaties R1 (S,V), R2 (T,Z,S) en R3 (T,Z,D). De onderstreepte attributen vormen de (kandidaat)sleutels. Merk op dat R2 twee kandidaatsleutels heeft.

**6.7** Er zijn vier entiteittypen, namelijk Cursussen, Docenten, Plaatsen en Geplande cursussen. Hiervoor worden respectievelijk de relaties CURSUSSEN, DOCENTEN, PLAATSEN en PLA-CUR ingevoerd. De eerste twee entiteittypen geven geen problemen. Een plaats heeft een maximum aantal deelnemers als attribuut (MAX). Verder is er de planning van een cursus. Een geplande cursus heeft een plannummer, datum, aantal inschrijvingen en een minimum aantal inschrijvingen. Bovendien bestaan er 1:n associatietypen tussen aan de ene kant Geplande cursussen en aan de andere kant de Cursussen, Docenten en Plaatsen. Rest nog het n:m associatietype Cursus-Docent, dat aangeeft welke docenten welke cursussen kunnen geven (met het functioneel afhankelijke attribuut GESCHIKTHEID). De relatie CUR-DOC geeft dat associatietype weer.

```
CURSUSSEN   (CNO, CNAAM)
DOCENTEN    (DNO, NAAM, ADRES)
PLAATSEN    (PLAATS, MAX)
CUR-DOC     (CNO, DNO, GESCHIKTHEID)
PLA-CUR     (PNO, CNO, DNO, PLAATS, N-INSCH, MIN, DATUM)
```

**6.8** Er zijn de volgende entiteittypen: Studenten, Modules en Instituten met de bijbehorende relaties STUDENTEN, MODULES en INSTITUTEN. Er zijn twee n:m associatietypen tussen Studenten en Modules weergegeven door respectievelijk PAKKETTEN en TENTAMENS (met als attribuut BEOORDELING). Het attribuut STATUSPAKKET hangt functioneel van student af, dus wordt in de relatie STUDENTEN geplaatst. Instituten kennen het niet-atomaire kenmerk onderwijsadres, hetgeen aanleiding geeft tot de relatie INST-ADR. Tussen Instituten en Modules bestaat ook een n:m associatietype gerepresenteerd door INST-MOD (met als functioneel afhankelijk attribuut ONDERWIJS-ADRES, het adres waar het module door het instituut wordt gegeven). Als op een adres slechts één instituut onderwijs mag verzorgen dan hebben we de FA ONDERWIJS-ADRES -> INAAM. We nemen echter aan dat op een en hetzelfde onderwijsadres meerdere instituten modules kunnen onderwijzen. Aangezien elk tweetal (INAAM,ONDERWIJS-ADRES) van INST-MOD moet voorkomen in de relatie INST-ADR hebben we hier te maken met een samengestelde vreemde sleutel. Om dat te kunnen aangeven, wijzigen we de tot nu toe gebruikte notatie: de primaire sleutel wordt vet gedrukt en een vreemde sleutel onderstreept. Dit resulteert in de volgende relaties:

```
STUDENTEN     (SNAAM, ADRES, WOONPLAATS, STATUSPAKKET)
MODULES       (MCODE, ZWAARTE)
INSTITUTEN    (INAAM, SECR-ADRES)
INST-ADR      (INAAM, ONDERWIJS-ADRES)
PAKKETTEN     (SNAAM, MCODE)
TENTAMENS     (SNAAM, MCODE, BEOORDELING)
INST-MOD      (MCODE, INAAM, ONDERWIJS-ADRES)
```

## HOOFDSTUK 7

Bedenk dat op de meeste vragen meer dan één antwoord mogelijk is.

**7.1**
```
SELECT   PNO, PNAAM, STAD
FROM     P
```

**7.2**
```
SELECT   PNO, PNAAM, STAD
FROM     P
WHERE    STAD = 'Londen'
```

**7.3**
```
SELECT   DISTINCT LNO
FROM     LAP
WHERE    PNO = 'P1'
```

**7.4**

```
SELECT   LNO, ANO, PNO, HOEV
FROM     LAP
WHERE    HOEV >= 300
AND      HOEV <= 750
```

**7.5**

```
SELECT   DISTINCT KLEUR, PLAATS
FROM     A
```

**7.6**

```
SELECT   LNO, ANO, PNO
FROM     L, A, P
WHERE    WNPL = PLAATS
AND      PLAATS = STAD
```

**7.7**

```
SELECT   LNO, ANO, PNO
FROM     L, A, P
WHERE    NOT
       ( WNPL = PLAATS AND PLAATS = STAD)
```

of

```
SELECT   LNO, ANO, PNO
FROM     L, A , P
WHERE    WNPL ~= PLAATS
OR       PLAATS ~= STAD
```

**7.8**

```
SELECT   LNO, ANO, PNO
FROM     L, A, P
WHERE    WNPL ~= PLAATS
AND      PLAATS ~= STAD
AND      STAD ~= WNPL
```

**7.9**

```
SELECT   DISTINCT ANO
FROM     LAP, L
WHERE    LAP.LNO = L.LNO
AND      WNPL = 'Londen'
```

**7.10**

```
SELECT   DISTINCT ANO
FROM     LAP, L, P
WHERE    LAP.LNO = L.LNO
AND      LAP.PNO = P.PNO
AND      WNPL = 'Londen'
AND      STAD = 'Londen'
```

**7.11**

```
SELECT   DISTINCT WNPL, STAD
FROM     L, LAP, P
WHERE    L.LNO = LAP.LNO
AND      LAP.PNO = P.PNO
```

**7.12**
```
SELECT   DISTINCT ANO
FROM     LAP, L, P
WHERE    LAP.LNO = L.LNO
AND      LAP.PNO = P.PNO
AND      WNPL = STAD
```

**7.13**
```
SELECT   DISTINCT P.PNO
FROM     LAP, L, P
WHERE    LAP.LNO = L.LNO
AND      LAP.PNO = P.PNO
AND      WNPL ~= STAD
```

**7.14**
```
SELECT   LAPX.ANO, LAPY.ANO
FROM     LAP LAPX, LAP LAPY
WHERE    LAPX.LNO = LAPY.LNO
AND      LAPX.ANO > LAPY.ANO
```

**7.15**
```
SELECT   COUNT (DISTINCT PNO)
FROM     LAP
WHERE    LNO = 'L1'
```

**7.16**
```
SELECT   SUM (HOEV)
FROM     LAP
WHERE    ANO = 'A1'
AND      LNO = 'L1'
```

**7.17**
```
SELECT   ANO, PNO, SUM(HOEV)
FROM     LAP
GROUP    BY ANO, PNO
```

**7.18**
```
SELECT   DISTINCT ANO
FROM     LAP
GROUP    BY ANO, PNO
HAVING   AVG (HOEV) > 320
```

**7.19**
```
SELECT   LNO, ANO, PNO, HOEV
FROM     LAP
WHERE    HOEV IS NOT NULL
```

**7.20**
```
SELECT   COUNT (*)
FROM     LAP
WHERE    HOEV IS NOT NULL
```

**7.21**

```
SELECT   PNAAM
FROM     P
WHERE    PNO IN
        (SELECT   PNO
         FROM     LAP
         WHERE    LNO = 'L1' )
```

**7.22**

```
SELECT   DISTINCT KLEUR
FROM     A
WHERE    ANO IN
        (SELECT   ANO
         FROM     LAP
         WHERE    LNO = 'L1')
```

**7.23**

```
SELECT   DISTINCT ANO
FROM     LAP
WHERE    PNO IN
        (SELECT   PNO
         FROM     P
         WHERE    STAD = 'Londen')
```

**7.24**

```
SELECT   DISTINCT PNO
FROM     LAP
WHERE    ANO IN
        (SELECT   ANO
         FROM     LAP
         WHERE    LNO = 'L1')
```

**7.25**

```
SELECT   DISTINCT LNO
FROM     LAP
WHERE    ANO IN
        (SELECT   ANO
         FROM     LAP
         WHERE    LNO IN
                 (SELECT   LNO
                  FROM     LAP
                  WHERE    ANO IN
                          (SELECT  ANO
                           FROM    A
                           WHERE   KLEUR='Rood')))
```

**7.26**

```
SELECT   LNO
FROM     L
WHERE    STATUS <
        (SELECT   STATUS
         FROM     L
         WHERE    LNO = 'L1')
```

**7.27**
```
SELECT   PNO
FROM     P
WHERE    STAD =
        (SELECT   MIN (STAD)
         FROM     P )
```

**7.28**
```
SELECT   PNO
FROM     LAP
WHERE    ANO = 'A1'
GROUP    BY PNO
HAVING   AVG (HOEV) >
        (SELECT   MAX (HOEV)
         FROM     LAP
         WHERE    PNO = 'P1' )
```

**7.29**
```
SELECT   DISTINCT LNO
FROM     LAP LAPX
WHERE    ANO = 'A1'
AND      HOEV >
        (SELECT   AVG (HOEV)
         FROM     LAP LAPY
         WHERE    ANO = 'A1'
         AND      LAPY.PNO = LAPX.PNO)
```

**7.30**
```
SELECT   DISTINCT ANO
FROM     LAP
WHERE    EXISTS
        (SELECT   *
         FROM     P
         WHERE    PNO = LAP.PNO
         AND      STAD = 'Londen')
```

**7.31**
```
SELECT   DISTINCT LAPX.PNO
FROM     LAP LAPX
WHERE    EXISTS
        (SELECT   *
         FROM     LAP LAPY
         WHERE    LAPY.ANO = LAPX.ANO
         AND      LAPY.LNO = 'L1' )
```

**7.32**
```
SELECT   PNO
FROM     P
WHERE    NOT EXISTS
        (SELECT   *
         FROM     LAP
         WHERE    PNO = P.PNO
```

```
                AND       ANO IN
                         (SELECT   ANO
                          FROM     A
                          WHERE    KLEUR = 'Rood')
                AND       LNO IN
                         (SELECT   LNO
                          FROM     L
                          WHERE    WNPL = 'Londen'   )
```

Anders gezegd:
"Geef de projecten PNO waarvoor niet een levering LAP bestaat, die een rood artikel betreft en gedaan is door een Londense leverancier".

**7.33**

```
SELECT    DISTINCT PNO
FROM      LAP LAPX
WHERE     NOT EXISTS
         (SELECT   *
          FROM     LAP LAPY
          WHERE    LAPY.PNO = LAPX.PNO
          AND      LAPY.LNO ~= 'L1')
```

Anders gezegd:
"Geef de projecten PNO waarvoor niet een levering LAPY bestaat, die niet door L1 gedaan is".

**7.34**

```
SELECT    DISTINCT ANO
FROM      LAP LAPX
WHERE     NOT EXISTS
         (SELECT   PNO
          FROM     P
          WHERE    STAD = 'Londen'
          AND      NOT EXISTS
                  (SELECT   *
                   FROM     LAP LAPY
                   WHERE    LAPY.ANO = LAPX.ANO
                   AND      LAPY.PNO = P.PNO ))
```

Anders gezegd:
"Geef de artikelen ANO waarvoor geen Londens project PNO bestaat zonder een levering LAPY die ANO en PNO betreft".

**7.35**

```
SELECT    DISTINCT LNO
FROM      LAP LAPX
WHERE     EXISTS
         (SELECT  ANO
          FROM    LAP LAPY
          WHERE   NOT EXISTS
```

```
(SELECT  PNO
 FROM    P
 WHERE   NOT EXISTS
        (SELECT  *
         FROM    LAP LAPZ
         WHERE   LAPZ.LNO=LAPX.LNO
         AND     LAPZ.ANO=LAPY.ANO
         AND     LAPZ.PNO=P.PNO)))
```

Anders geformuleerd:
"Geef de leveranciers LNO waarvoor geldt dat er een artikel LAPY.ANO bestaat waarbij geen project P.PNO bestaat waaraan leverancier LNO niet artikel LAPY.ANO kan leveren".

Merk op dat "SELECT ANO FROM ..." wordt gebruikt waar ook "SELECT * FROM ..." gebruikt had kunnen worden: de laatste formulering is wel goed maar de eerste geeft beter weer waar het hier om gaat.

**7.36**

```
SELECT   DISTINCT PNO
FROM     LAP LAPX
WHERE    NOT EXISTS
        (SELECT   ANO
         FROM     LAP LAPY
         WHERE    LAPY.LNO = 'L1'
         AND      NOT EXISTS
                 (SELECT   *
                  FROM     LAP LAPZ
                  WHERE    LAPZ.ANO = LAPY.ANO
                  AND      LAPZ.PNO = LAPX.PNO))
```

**7.37**
"Geef de projecten LAPX.PNO waarvoor geldt dat ze geen artikel LAPY.ANO geleverd krijgen dat ook niet door L1 geleverd wordt".
Anders gezegd:
"Geef de nummers van die projecten die alleen artikelen gebruiken die leverancier L1 levert".

**7.38**
"Geef de projecten LAPX.PNO waarvoor geldt dat er geen door L1 geleverd artikel LAPY.ANO bestaat dat niet door L1 aan LAPX.PNO wordt geleverd".
Anders gezegd:
"Geef de nummers van die projecten waaraan leverancier L1 al zijn artikelen (zijn hele assortiment) levert".

**7.39**
"Geef de projecten LAPX.PNO waarvoor geldt dat er geen artikel LAPY.ANO geleverd aan LAPX.PNO bestaat dat niet door L1 aan LAPX.PNO wordt geleverd".
Anders gezegd:
"Geef de nummers van die projecten waarvoor geldt dat elk benodigd artikel (ook) door leverancier L1 wordt geleverd".

**7.40**
"Geef de projecten LAPX.PNO waarvoor geen leverancier LAPY.LNO bestaat die een rood artikel levert en die niet aan dat project LAPX.PNO levert".
Anders gezegd:
"Geef de nummers van die projecten waarvoor geldt dat elke leverancier van een rood artikel ook aan dat project levert".

**7.41**
```
UPDATE   A
SET      KLEUR = 'Oranje'
WHERE    KLEUR = 'Rood'
```

**7.42**
```
DELETE
FROM     P
WHERE    PNO NOT IN
        (SELECT   PNO
         FROM     LAP)
```

**7.43**
```
DELETE
FROM     LAP
WHERE    'Rome' =
        (SELECT   STAD
         FROM     P
         WHERE    P.PNO = LAP.PNO)

DELETE
FROM     P
WHERE    STAD = 'Rome'
```

**7.44**
```
INSERT
INTO     L (LNO, LNAAM, WNPL)
VALUES    ('L10', 'Witte', 'New York')
```
of
```
INSERT
INTO     L ( LNO, LNAAM, STATUS, WNPL )
VALUES    ('L10', 'Witte', NULL, 'New York')
```

**7.45**

```
CREATE TABLE LONDEN-ART
    ( ANO CHAR(6) )

INSERT
INTO     LONDEN-ART (ANO)
  SELECT   DISTINCT ANO
  FROM     LAP
  WHERE    LNO IN
          (SELECT   LNO
           FROM     L
           WHERE    WNPL = 'Londen')
  OR       PNO IN
          (SELECT   PNO
           FROM     P
           WHERE    STAD = 'Londen')
```

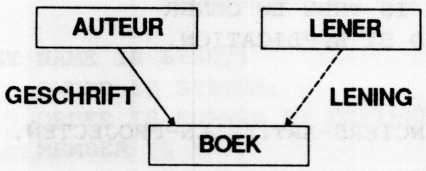
**7.46**

```
CREATE TABLE LONDEN-PROJ
    ( PNO  CHAR (4) )

INSERT
INTO     LONDEN-PROJ (PNO)
  SELECT   PNO
  FROM     P
  WHERE    STAD = 'Londen'
  OR       PNO IN
          (SELECT   PNO
           FROM     LAP
           WHERE    LNO IN
                   (SELECT   LNO
                    FROM     L
                    WHERE    WNPL = 'Londen'))
```

## HOOFDSTUK 8

**8.3**




a. Ja.
b. Neen.
c. Ja.

d. Ja.
e. Ja.
f. Ja.

**8.4** Het membership van LINK in het settype WL is AUTOMATIC, omdat een LINK record meteen na de creatie aan een (manager) record moet worden gekoppeld. Het membership van WERKN in LW is MANUAL, omdat tenminste één WERKN record, namelijk dat van de 'directeur', niet een member van een LW set is. Merk op dat het LINK record geen velden kent.

```
SCHEMA NAME IS MANAGEMENT.

RECORD NAME IS WERKN;
    KEY W-SLT IS WNO IN WERKN
        DUPLICATES ARE NOT ALLOWED.
    02  WNO      ; TYPE IS CHARACTER 5.
    02  WNAAM    ; TYPE IS CHARACTER 20.
    02  WADRES   ; TYPE IS CHARACTER 15.

RECORD NAME IS LINK;

SET NAME IS WL;
    OWNER IS WERKN;
    ORDER IS NEXT.
    MEMBER IS LINK;
        INSERTION IS AUTOMATIC
        RETENTION IS FIXED;
        SET SELECTION IS THRU WL OWNER
            IDENTIFIED BY KEY W-SLT.

SET NAME IS LW;
    OWNER IS LINK;
    ORDER IS NEXT.
    MEMBER IS WERKN
        INSERTION IS OPTIONAL
        RETENTION IS MANUAL
        SET SELECTION IS THRU LW OWNER
            IDENTIFIED BY APPLICATION.
```

**8.5**

```
SCHEMA NAME IS LEVERANCIERS-ARTIKELEN-PROJECTEN.

RECORD NAME IS L;
    KEY L-SLT IS LNO IN L
        DUPLICATES ARE NOT ALLOWED.
    02  LNO      ; TYPE IS CHARACTER 5.
    02  LNAAM    ; TYPE IS CHARACTER 20.
```

```
    02  STATUS  ; TYPE IS DECIMAL 3.
    02  WNPL    ; TYPE IS CHARACTER 15.

RECORD NAME IS A;
    KEY A-SLT IS ANO IN A
        DUPLICATES ARE NOT ALLOWED.
    02  ANO     ; TYPE IS CHARACTER 6.
    02  ANAAM   ; TYPE IS CHARACTER 20.
    02  KLEUR   ; TYPE IS CHARACTER 6.
    02  GEW     ; TYPE IS DECIMAL 6.
    02  PLAATS  ; TYPE IS CHARACTER 15.

RECORD NAME IS P
    KEY P-SLT IS PNO IN P
        DUPLICATES ARE NOT ALLOWED.
    02  PNO     ; TYPE IS CHARACTER 4.
    02  PNAAM   ; TYPE IS CHARACTER 20.
    02  STAD    ; TYPE IS CHARACTER 15.

RECORD NAME IS LAP;
    KEY LAP-SLT IS LNO IN LAP, ANO IN LAP, PNO IN LAP
        DUPLICATES ARE NOT ALLOWED.
    02  LNO     ; TYPE IS CHARACTER 5.
    02  ANO     ; TYPE IS CHARACTER 6.
    02  PNO     ; TYPE IS CHARACTER 4.
    02  HOEV    ; TYPE IS DECIMAL 5.

SET NAME IS L-LAP;
    OWNER IS L;
    ORDER IS SORTED BY DEFINED KEYS.
    MEMBER IS LAP;
        INSERTION IS AUTOMATIC
        RETENTION IS FIXED;
        KEY IS ASCENDING ANO IN LAP, PNO IN LAP
            DUPLICATES ARE NOT ALLOWED
        SET SELECTION IS THRU L-LAP OWNER
            IDENTIFIED BY KEY L-SLT.

SET NAME IS L-SET;
    OWNER IS SYSTEM;
    ORDER IS SORTED BY DEFINED KEYS.
    MEMBER IS L;
        INSERTION IS AUTOMATIC
        RETENTION IS FIXED;
        KEY IS ASCENDING LNO IN L.
```

```
SET NAME IS A-LAP;
    OWNER IS A;
    ORDER IS SORTED BY DEFINED KEYS.
    MEMBER IS LAP;
        INSERTION IS AUTOMATIC
        RETENTION IS FIXED;
        KEY IS ASCENDING PNO IN LAP, LNO IN LAP
            DUPLICATES ARE NOT ALLOWED
        SET SELECTION IS THRU A-LAP OWNER
            IDENTIFIED BY KEY A-SLT.

SET NAME IS A-SET;
    OWNER IS SYSTEM;
    ORDER IS SORTED BY DEFINED KEYS.
    MEMBER IS A;
        INSERTION IS AUTOMATIC
        RETENTION IS FIXED;
        KEY IS ASCENDING ANO IN A.

SET NAME IS P-LAP;
    OWNER IS P;
    ORDER IS SORTED BY DEFINED KEYS.
    MEMBER IS LAP;
        INSERTION IS AUTOMATIC
        RETENTION IS FIXED;
        KEY IS ASCENDING LNO IN LAP, ANO IN LAP
            DUPLICATES ARE NOT ALLOWED
        SET SELECTION IS THRU P-LAP OWNER
            IDENTIFIED BY KEY P-SLT.

SET NAME IS P-SET;
    OWNER IS SYSTEM;
    ORDER IS SORTED BY DEFINED KEYS.
    MEMBER IS P;
        INSERTION IS AUTOMATIC
        RETENTION IS FIXED;
        KEY IS ASCENDING PNO IN P.
```

**8.6**

```
SCHEMA NAME IS ARTIKELEN.

RECORD NAME IS ARTIKEL;
    KEY ART-SLT IS ANO IN ARTIKEL
        DUPLICATES ARE NOT ALLOWED.
    02  ANO      ; TYPE IS CHARACTER 5.
    ....
```

```
RECORD NAME IS LINK;
    02  HOEV     ; TYPE IS DECIMAL 5.
    ....
SET NAME IS ME;
    OWNER IS ARTIKEL;
    ORDER IS NEXT.
    MEMBER IS LINK;
        INSERTION IS AUTOMATIC
        RETENTION IS FIXED
        SET SELECTION IS THRU ME OWNER
            IDENTIFIED BY KEY ART-SLT.

SET NAME IS WG;
    OWNER IS ARTIKEL;
    ORDER IS NEXT.
    MEMBER IS LINK
        INSERTION IS AUTOMATIC
        RETENTION IS FIXED
        SET SELECTION IS THRU WG OWNER
            IDENTIFIED BY KEY ART-SLT.
```

**8.7** Een aantal observaties kunnen hierbij behulpzaam zijn. Als O een bestaand recordtype is (dat wil zeggen 'oud' is), dan zal ieder programma dat ERASE opdrachten uitvoert met betrekking tot O *mogelijk* gewijzigd moeten worden (tezamen met het corresponderende subschema). Als M oud is en de membership class van OM is AUTOMATIC, dan zal elk programma dat nu STORE opdrachten uitvoert met betrekking tot M gewijzigd moeten worden (en tevens het bijbehorende subschema).

## HOOFDSTUK 9

**9.1** OEUVRE en SCHRIJVERS zijn beide AUTOMATIC/FIXED settypen.

**9.2**

**a.** De entiteittypen Auteurs, Tijdschriften, Artikelen en Onderwerpen resulteren elk in een recordtype. Het m:n associatietype tussen Artikelen en Tijdschriften wordt een recordtype ART-TIJDSCH; hetzelfde geldt voor het m:n associatietype dat bestaat tussen Auteurs en Artikelen. Het bijbehorende recordtype heet AUT-ART. Beide m:n associatietypen geven aanleiding tot twee settypen. De 1:n associatietypen worden omgezet in een settype, terwijl het niet-atomaire kenmerk kernwoorden door een data aggregaat kan worden gerepresenteerd:

```
RECORD NAME IS ONDERWERPEN;
    KEY OND-SLT IS ONDERW-NO IN ONDERWERPEN
        DUPLICATES ARE NOT ALLOWED.
    02  ONDERW-NO  ....

RECORD NAME IS ARTIKELEN;
    KEY ART-SLT IS TITEL IN ARTIKELEN
        DUPLICATES ARE NOT ALLOWED.
    02  TITEL  ....
    02  KERNWRD   OCCURS 10 TIMES.
        03  KERNWOORD  .....

RECORD NAME IS AUTEURS;
    KEY A-SLT IS ANAAM IN AUTEURS
        DUPLICATES ARE NOT ALLOWED.
    02 ANAAM  ....

RECORD NAME IS TIJDSCHRIFTEN;
    KEY TNAAM-SLT IS TNAAM IN TIJDSCHRIFTEN
        DUPLICATES ARE NOT ALLOWED.
    02 TNAAM  ....

RECORD NAME IS ART-TIJDSCH;
    KEY AT-SLT IS TITEL IN ART-TIJDSCH,
                  TNAAM IN ART-TIJDSCH
        DUPLICATES ARE NOT ALLOWED.
    02 TITEL  ....
    02 TNAAM  ....

RECORD NAME IS AUT-ART;
    KEY AA-SLT IS ANAAM IN AUT-ART,
                  TITEL IN AUT-ART
        DUPLICATES ARE NOT ALLOWED.
    02 ANAAM  ....
    02 TITEL  ....

SET NAME IS VERSCHIJNT-IN
    ...
    OWNER IS ARTIKEL  ...
    MEMBER IS ART-TIJDSCH  ...

SET NAME IS INHOUD
    ...
    OWNER IS TIJDSCHRIFT  ...
    MEMBER IS ART-TIJDSCH  ...
```

```
SET NAME IS WERK
    ...
    OWNER IS AUTEURS  ...
    MEMBER IS AUT-ART  ...

SET NAME IS SCHRIJV
    ...
    OWNER IS ARTIKELEN  ...
    MEMBER IS AUT-ART  ...

SET NAME IS KLASSE
    ...
    OWNER IS ONDERWERPEN  ...
    MEMBER IS ARTIKELEN  ...
```

**b.** Men kan in de orders database voor ORDERREGELS een apart recordtype (met bijbehorend settype) invoeren. Dat wordt meestal gedaan, doch hier niet.

```
RECORD NAME IS KLANTEN;
    KEY K-SLT IS KNAAM IN KLANTEN
        DUPLICATES ARE NOT ALLOWED.
    02  KNAAM  ....

RECORD NAME IS ARTIKELEN;
    KEY ART-SLT IS ANAAM IN ARTIKELEN
        DUPLICATES ARE NOT ALLOWED.
    02  ANAAM  ....

RECORD NAME IS ORDERS;
    KEY ORD-SLT IS ORDERNO IN ORDERS
        DUPLICATES ARE NOT ALLOWED.

    02  ORDERNO  ....
    02  KNAAM  .....
    02  ....
    02  REGELS   OCCURS 10 TIMES.
        03  REGELNO  ....
        03  ANAAM  ....
        03  HOEVEELHEID  ....

SET NAME IS BESTELLINGEN
    OWNER IS KLANTEN  ...
    MEMBER IS ORDERS  ...
```

```
SET NAME IS ART-BEHOEFTE
    OWNER IS ARTIKELEN  ...
    MEMBER IS ORDERS  ...
```

**c.** De studenten database:

```
RECORD NAME IS PERSONEN;
    KEY PERS-SLT IS PNAAM IN PERSONEN
        DUPLICATES ARE NOT ALLOWED.
    02  PNAAM  ....

RECORD NAME IS DOCENTEN;
    KEY DOC-SLT IS DNAAM IN DOCENTEN
        DUPLICATES ARE NOT ALLOWED.
    02  DNAAM  ....

RECORD NAME IS STUDENTEN;
    KEY STU-SLT IS SNAAM IN STUDENTEN
        DUPLICATES ARE NOT ALLOWED.
    02  SNAAM  ....

RECORD NAME IS COLLEGES;
    KEY COL-SLT IS CNO IN COLLEGES
        DUPLICATES ARE NOT ALLOWED.
    02  CNO  ....

RECORD NAME IS STUD-COLL;
    KEY SC-SLT IS CNO IN STUD-COLL,
                  SNAAM IN STUD-COLL
        DUPLICATES ARE NOT ALLOWED.
    02  CNO  ....
    02  SNAAM  ....

SET NAME IS SPECIALIS-DOC   (* specialisatie docent *)
    ...
    OWNER IS PERSONEN  ...
    MEMBER IS DOCENTEN  ...

SET NAME IS SPECIALIS-STUD  (* idem student *)
    ...
    OWNER IS PERSONEN  ...
    MEMBER IS STUDENTEN  ...

SET NAME IS DOC-COLLEG
    ...
    OWNER IS DOCENTEN  ...
    MEMBER IS COLLEGES  ...
```

```
SET NAME IS VAKKEN
    ...
    OWNER IS STUDENTEN ...
    MEMBER IS STUD-COLL ...

SET NAME IS TOEHOORDERS
    ...
    OWNER IS COLLEGES ...
    MEMBER IS STUD-COLL ...
```

## HOOFDSTUK 10

**10.1** Merk op dat de members van een L-LA set op ANO gesorteerd zijn.

| Na uitvoering van opdracht | (1) | (2) | (3) |
|---|---|---|---|
| Current van run-unit | 'L2' | 'L2/A1' | 'L2/A2' |
| Current L record | 'L2' | 'L2' | 'L2' |
| Current A record | -- | -- | -- |
| Current LA record | -- | -- | 'L2/A2' |
| Current record van settype L-LA | 'L2' | 'L2/A1' | 'L2/A2' |
| Current record van settype A-LA | -- | 'L2/A1' | 'L2/A1' |
| Current L-LA set; owner: | 'L2' | 'L2' | 'L2' |
| Current A-LA set; owner: | -- | 'A1' | 'A2' |
| Current record van realm L-LA-A | 'L2' | 'L2/A1' | 'L2/A2' |

**10.2** We nemen aan dat er declaratieve procedures (zoals geschetst in paragraaf 10.3) aanwezig zijn om uitzonderlijke situaties op te vangen.

**a.**

```
READY   USAGE-MODE IS SHARED RETRIEVAL
MOVE    'NO' TO NOTFOUND
MOVE    'P1' TO PNO IN P
FIND    ANY P USING PNO IN P
IF      NOTFOUND = 'NO'
   MOVE    'NO' TO EOF
   FIND    FIRST LAP WITHIN P-LAP
```

```
    PERFORM  UNTIL EOF = 'YES'
       GET   LAP
       (* voeg LNO IN LAP toe aan resultaat
          tenzij al aanwezig *)
       FIND  NEXT LAP WITHIN P-LAP
    END-PERFORM
END-IF
FINISH
```

**b.**

```
READY   USAGE-MODE IS SHARED RETRIEVAL
MOVE    'NO' TO EOF
MOVE    'P1' TO PNO IN P
MOVE    'A1' TO ANO IN LAP
FIND    LAP WITHIN P-LAP USING ANO IN LAP
PERFORM   UNTIL EOF = 'YES'
   GET    LAP
   (* voeg LNO IN LAP toe aan resultaat *)
   FIND   DUPLICATE WITHIN P-LAP USING ANO IN LAP
END-PERFORM
FINISH
```

**c.**

```
READY   USAGE-MODE IS SHARED RETRIEVAL
MOVE    'NO' TO NOTFOUND
MOVE    'P1' TO PNO IN P
FIND    ANY P USING PNO IN P
IF   NOTFOUND = 'NO'
     MOVE    'NO' TO EOF
     FIND    FIRST LAP WITHIN P-LAP
     PERFORM   UNTIL EOF = 'YES'
       GET     LAP
       FIND    OWNER IN A-LAP
       GET     A
       IF      KLEUR IN A = 'Rood'
         (* voeg LNO IN LAP toe aan resultaat
           tenzij al aanwezig *)
       END-IF
     END-PERFORM
FINISH
```

**d.**

```
READY   SHARED RETRIEVAL
MOVE    'NO' TO EOF
FIND    FIRST P IN P-SET
PERFORM   UNTIL EOF = 'YES'
   GET     P
   MOVE    'NO' TO EOF
   MOVE    'NO' TO BADP
   FIND    FIRST LAP WITHIN P-LAP
```

```
PERFORM   UNTIL EOF = 'YES' OR BADP = 'YES'
   FIND   OWNER WITHIN A-LAP
   GET    A
   IF     KLEUR IN A ='Rood'
          MOVE   'YES' TO BADP
   ELSE
      FIND  OWNER WITHIN L-LAP
      GET   L
      IF    WNPL IN L = 'Londen'
         MOVE  'YES' TO BADP
      ELSE
         FIND  NEXT LAP WITHIN P-LAP
      END-IF
END-PERFORM
IF     EOF = 'YES'
   MOVE    'NO' TO EOF
   (* voeg PNO IN P aan resultaat toe *)
END-IF
FIND    NEXT P WITHIN P-SET
END-PERFORM
FINISH
```

**e.** We gebruiken een keep list, KX, van lengte 2. Onze oplossing wordt efficiënter als de duplicaten in plaats van op het eind, zo spoedig mogelijk worden geëlimineerd.

```
LD    KX LIMIT IS 2.

READY   SHARED RETRIEVAL
MOVE    'NO' TO NOTFOUND
MOVE    'L1' TO LNO IN L
FIND    ANY L USING LNO IN L
IF  NOTFOUND = 'NO'
   MOVE   'NO' TO EOF
   FIND   FIRST LAP WITHIN L-LAP
   PERFORM  UNTIL EOF = 'YES'
      KEEP   USING KX
      FIND   OWNER WITHIN A-LAP
      FIND   FIRST LAP WITHIN A-LAP
      PERFORM  UNTIL EOF = 'YES'
         KEEP USING KX
         FIND  OWNER WITHIN L-LAP
         FIND  FIRST LAP WITHIN A-LAP
         PERFORM  UNTIL EOF = 'YES'
            GET   LAP
            (* voeg ANO IN LAP toe aan resultaat
             tenzij al aanwezig *)
```

```
            FIND   NEXT LAP WITHIN L-LAP
          END-PERFORM
          MOVE    'NO' TO EOF
          FIND   LAST WITHIN KX
          FREE   LAST WITHIN KX
          FIND   NEXT LAP WITHIN A-LAP
        END-PERFORM
        MOVE    'NO' TO EOF
        FIND   LAST WITHIN KX
        FREE   LAST WITHIN KX
        FIND   NEXT LAP WITHIN L-LAP
      END-PERFORM
    END-IF
    FINISH
```

**f.**

```
READY   SHARED RETRIEVAL
MOVE    'NO' TO EOF
FIND    FIRST L WITHIN L-SET
PERFORM  UNTIL EOF = 'YES'
  GET    L
  FIND   FIRST LAP WITHIN L-LAP
  PERFORM  UNTIL EOF = 'YES'
    FIND   OWNER IN P-LAP
    GET    P
    (* voeg <WNPL,STAD> toe aan resultaat
     tenzij al aanwezig *)
    FIND   NEXT LAP WITHIN L-LAP
  END-PERFORM
  MOVE    'NO' TO EOF
  FIND   NEXT L WITHIN L-SET
END-PERFORM
FINISH
```

**g.** Het is erg moeilijk om dit probleem efficiënt op te lossen. Onze oplossing bepaalt één voor één alle A records. Dit lijkt een niet zo wenselijke oplossing. Echter, de Format 2 FIND kan niet worden gebruikt (waarom niet?) en de Format 6 FIND is nog minder efficiënt (waarom?). (Tip: Om in te zien waarom de Format 2 FIND niet kan worden gebruikt, beschouw de situatie waarin de rode artikelen niet werden gewijzigd maar werden verwijderd.) De COMMIT op het eind is nodig omdat er een bijwerkoperatie is gedaan.

```
READY   PROTECTED UPDATE
MOVE    'NO' TO EOF
FIND    FIRST A WITHIN A-SET
PERFORM  UNTIL EOF = 'YES'
  GET    KLEUR IN A
```

```
    IF  KLEUR IN A = 'Rood'
        MOVE   'Oranje' TO KLEUR IN A
        MODIFY KLEUR IN A
    END-IF
    FIND   NEXT A WITHIN A-SET
END-PERFORM
COMMIT
FINISH
```

**h.** We nemen aan dat er een LAP record voor L1/A1/P1 bestaat; we nemen echter niet aan dat een LAP record voor L2/A1/P1 niet bestaat. De eerste stap is daarom het LAP record (als het bestaat) behorend bij L2/A1/P1 op te sporen (via een FIND) en te lezen (via een GET), zodat de HOEV waarde kan worden opgeteld bij de HOEV waarde van L1/A1/P1 om de nieuwe HOEV waarde van L2/A1/P1 te worden. Als het betreffende LAP record niet bestaat, dan hoeft er niets bij de HOEV waarde van L1/A1/P1 geteld te worden. We verwijderen eveneens het LAP record (als het bestaat) dat hoort bij L2/A1/P1.

```
READY   EXCLUSIVE UPDATE
MOVE    'NO' TO EOF
MOVE    'L2' TO LNO IN L
MOVE    'A1' TO ANO IN LAP
MOVE    'P1' TO PNO IN LAP
FIND    LAP WITHIN L-LAP USING ANO IN LAP, PNO IN LAP
IF  EOF = 'NO'
    GET     HOEV IN LAP
    MOVE    HOEV IN LAP TO TEMPHOEV
    ERASE   LAP
ELSE
    MOVE    'NO' TO EOF
    MOVE    ZERO TO TEMPHOEV
END-IF
```

We gaan nu verder met het LAP record dat hoort bij L1/A1/P1. Dit record wordt gelokaliseerd, (FIND) gewijzigd (MODIFY) en uit een set gehaald en weer in een andere set geplaatst (RECONNECT).

```
MOVE    'L1' TO LNO IN L
FIND    LAP WITHIN L-LAP USING ANO IN LAP, PNO IN LAP
GET     HOEV IN LAP
ADD     TEMPHOEV TO HOEV IN LAP
MOVE    'L2' TO LNO IN LAP
MODIFY  LNO IN LAP, HOEV IN LAP
MOVE    'L2' TO LNO IN L
```

```
RECONNECT  LAP WITHIN L-LAP
COMMIT
FINISH
```

**10.3**

**a.**

```
READY   SHARED RETRIEVAL
MOVE    'NO' TO EOF
FIND    FIRST ARTIKEL WITHIN ARTIKEL-SET
PERFORM  UNTIL EOF = 'YES'
   GET     ARTIKEL
   (* druk ANO IN ARTIKEL af *)
   FIND    FIRST LINK WITHIN ME
   PERFORM  UNTIL EOF = 'YES'
      FIND    OWNER WITHIN WG
              RETAINING ARTIKEL-SET, ME CURRENCNY
      GET     ARTIKEL
      (* druk ANO IN ARTIKEL, dwz componentnr af *)
      FIND    NEXT LINK WITHIN ME
   END-PERFORM
   MOVE    'NO' TO EOF
   FIND    CURRENT ARTIKEL WITHIN ARTIKEL-SET
   FIND    FIRST LINK WITHIN WG
   PERFORM  UNTIL EOF = 'YES'
      FIND    OWNER WITHIN ME
              RETAINING ARTIKEL-SET, WG CURRENCY
      GET     ARTIKEL
      (* druk ANO IN ARTIKEL, dwz assembly nr af *)
      FIND    NEXT ARTIKEL WITHIN WG
   END-PERFORM
   MOVE    'NO' TO EOF
   FIND    NEXT ARTIKEL WITHIN ARTIKEL-SET
END-PERFORM
FINISH
```

**b.**

```
LD    STACK LIMIT IS 10.  (* grens is willekeurig *)

READY   SHARED RETRIEVAL
MOVE    'NO' TO NOTFOUND
MOVE    GIVEN-ANO TO ANO IN ARTIKEL
FIND    ANY ARTIKEL USING ANO IN ARTIKEL
IF  NOTFOUND = 'NO'
    GET     ARTIKEL
    (* druk af ANO IN ARTIKEL *)
    MOVE    ZERO TO STACK-DEPTH
    MOVE    'NO' TO EOF
    PERFORM  UNTIL STACK-DEPTH < ZERO
       FIND    NEXT LINK WITHIN ME
```

```
        IF  EOF = 'NO'
            GET     LINK
            KEEP    USING STACK
            ADD     1 TO STACK-DEPTH
            FIND    OWNER WITHIN WG
            GET     ARTIKEL
            (* druk ANO IN ARTIKEL af *)
        ELSE
            IF  STACK-DEPTH > ZERO
                FIND    LAST WITHIN STACK
                FREE    LAST WITHIN STACK
                MOVE    'NO' TO EOF
            END-IF
            SUBTRACT    1 FROM STACK-DEPTH
        END-IF
    END-PERFORM
END-IF
FINISH
```

## HOOFDSTUK 11

**11.1** Via de WNPL index worden de RID's van de Londense leveranciers gevonden en een RID geeft het paginanummer.

**11.2** Bij een bestand van N cilinders is het gemiddelde aantal cilinders dat bij een willekeurige seek overbrugd moet worden (1/3) * N (toon dat aan).

De afmeting van het bestand is 60 cilinders; dus als er geen onderbreking bestaat, dan is de gemiddelde seekafstand (1/3) * 60 = 20 cilinders. Wat is de kans dat de onderbreking van 10 cilinders moet worden overgestoken?

- Het begin van de seek is in (1/3) van de gevallen in het gebied 20...39.
- Het einde van de seek is in (2/3) van de gevallen in het gebied 50...89.

Dat wil zeggen dat de kans 2/9 is om in het gebied 20...39 te beginnen en in 50...89 te eindigen.
Andersom geldt iets dergelijks. Dus in (4/9) van de gevallen komt er nog eens een 'extra' overbrugging van 10 cilinders bij.
In totaal is de gemiddelde seek:

20 + (4/9) * 10 = 24.45 cilinders.

**11.3** De voordelen van indexen ten opzichte van heaps zijn:
**a.** Zij versnellen directe toegang gebaseerd op een gegeven waarde van het geïndexeerde veld. Zonder de index is het doorzoeken van het hele bestand nodig.

**b.** Zij versnellen de afhandeling van vragen waarbij een rij records gesorteerd op het geïndexeerde veld moeten worden afgeleverd. Zonder de index is een extra sorteerslag nodig.

De nadelen zijn de volgende:
**a.** Indexen beslaan ruimte op schijf.
**b.** De bijwerkoperaties worden vertraagd.

**11.4** Als een record wordt verwijderd, dan kan dit op twee manieren gebeuren.

De eerste manier bestaat eruit dat een 'verwijderindicatie' in het record wordt aangebracht. Met dit alternatief ziet een leesoperatie er als volgt uit. Na het hashen wordt de plek op schijf onderzocht waar het record had moeten staan. Als het record er niet staat (ten gevolge van collisie), dan wordt verder gezocht totdat een vrije plaats of het gezochte record is gevonden. In het eerste geval is het gezochte record niet aanwezig.

De tweede manier maakt de ruimte die bezet wordt door het te verwijderen record vrij. Nu moet bij een leesoperatie het hele bestand afgezocht worden voordat zeker is dat het record er niet is (Waarom?).

**11.5**
**a.** Sorteren heeft zin, omdat voor een niet aanwezig record gemiddeld maar de helft van het bestand doorzocht hoeft te worden. Bij een niet gesorteerd bestand zal men tot aan het eind moeten doorzoeken.
**b.** Nu heeft sorteren geen zin: gemiddeld moet de helft van het bestand doorzocht worden.
**c.** Sorteren heeft zin als de groep sleutelwaarden ook gesorteerd wordt. Door één keer door het hele bestand te lopen, kunnen alle vragen nu een voor een beantwoord worden.

**11.6** De zogenaamde index-sequentiële bestandsorganisatie ziet er als volgt uit:
- Het bestand wordt gesorteerd op sleutel en op de beschikbare sporen en cilinders opgeslagen.
- Per cilinder is er een zogenaamde spoorindextabel die wordt opgeslagen in het eerste blok van een cilinder en die per spoor de hoogste sleutelwaarde van de op dat spoor opgeslagen records bevat.
- Bovendien is er een zogenaamde cilinderindextabel die per cilinder de hoogste sleutelwaarde geeft van alle records die op die cilinder zijn opgeslagen.
- Om invoegingen aan te kunnen, is het nodig een overloopgebied te definiëren. Meestal is er per cilinder een overloopgebied (men wijst bijvoorbeeld het laatste spoor van een cilinder als overloopgebied aan). Ook dat gebied kan echter vollopen. Dus moet er per bestand ook nog

een overloopgebied zijn waar de records naar toegaan die niet op een bepaalde cilinder opgeslagen kunnen worden.

**11.7** Hashing en indexering komen beide in aanmerking. Aangezien de directe toegang bij hashing sneller is en omdat er geen sequentiële toegang op ANAAM gewenst is, is hashing te prefereren.

# Literatuur


[ANSI75] ANSI/X3/SPARC Study Group on Data Base Management Systems. Interim Report. FDT (ACM SIGMOD bulletin) 7, No 2, 1975.

[BACH69] C.W. Bachman: "Data Structure diagrams", Data Base, Vol 1, No 2, 1969.[BEHG87] P.A. Bernstein, V. Hadzilacos, N. Goodman: "Concurrency Control and Recovery in Database Systems", Addison-Wesley Publishing Company, Reading, Mass., 1987.

[BEHG87] P.A. Bernstein, V. Hadzilacos, N. Goodman: "Concurrency Control and Recovery in Database Systems", Addison-Wesley Publishing Company, Reading, Mass., 1987.

[CHAM76] D.D. Chamberlin e.a.: "SEQUEL 2: A unified approach to Data Definition. Manipulation and Control", IBM Research & Development, Volume 20, No 6, November 1976, pp 560-575.

[CHEN76] P.P.-S. Chen: "The Entity-Relationship Model-Toward a Unified View of Data." ACM Transactions on database Systems, Vol 1, No 1, Maart 1976, pp 9-36.

[CODA71] Data Base Task Group of CODASYL Programming Language Commmittee. Report, April 1971. Beschikbaar via ACM.

[CODD70] E.F. Codd: "A Relational Model of Data for Large Shared Banks", Communications of the ACM, Vol 13, No 6, Juni 1970, pp 377-387.

[CODD79] E.F. Codd: "Extending the Database Relational Model to Capture more Meaning", ACM Transactions on Database Systems, Vol 4, No 4, 1979, pp 397-434.

[DATE81] C.J. Date: "An Introduction to Database Systems", Vol 1, derde editie, Addison-Wesley Publishing Company, Reading, Mass., 1981.

[DATE83] C.J. Date: "An Introduction to Database Systems", Vol 2, Addison-Wesley Publishing Company, Reading Mass., 1983,

[DATE84] C.J. Date: "A Guide to DB2", Addison-Wesley Publishing Company, Reading, Mass., 1984.

[DATE86] C.J. Date: "An Introduction to Database Systems", Vol 1, vierde editie, Addison-Wesley Publishing Company, Reading Mass., 1986.

[HEAT71] I.J. Heath: "Unacceptable File Operations in a Relational Database", Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control, Nov. 1971.

[KLUW87] Codasyl and COBOL: Data Base Reports 1981, Kluwer, Deventer, 1987.

[RISS77] J. Rissanen: "Independent Components of Relations", ACM Transactions on Database Systems 2, No 4, December 1977, pp 319-325.

[TEFR82] T.J. Teorey en J.P. Fry: "Design of Database Structures", Prentice-Hall, Englewood Cliffs, N.J., 1982.

[TSKL78] D.C. Tsichritzis en A. Klug (eds): "The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Data Base Management Systems", Information Systems, No 3, 1978.

[TSLO82] D.C. Tsichritzis en H. Lochovsky: "Data Models", Prentice-Hall, Englewood Cliffs, N.J., 1982.

[VDLA87] R. van de Lans: "Het SQL Leerboek", tweede druk, Academic Service, Den Haag, 1987.

# Appendix A
# SUBSET SQL

Hieronder wordt de syntax van de in dit boek behandelde subset van de SQL-DML behandeld. Voor een volledige syntax zie [VDLA87].
De symbolen "[", "]", "{", "}", "|" en "..." zijn metasymbolen; dat zijn symbolen die gebruikt worden om de syntax van een taal te beschrijven.

1. Tekst tussen twee *vierkante haken* moet nul of één keer voorkomen.
   [ A ]  A is optioneel.

2. Een drietal punten "..." onmiddellijk volgend op een term tussen twee vierkante haken betekent dat de term tussen de vierkante haken *nul of meer keren* moet worden ingevuld en dat eventuele herhalingen moeten worden gescheiden door een komma.
   [ A ] ...  Bijvoorbeeld een lege string.

3. Het teken | scheidt alternatieven. Uit een rijtje alternatieven omgeven door accolades moet één alternatief worden gekozen.
   { A | B }  Hier moet òf A òf B worden ingevuld.

4. Een drietal punten "..." onmiddellijk volgend op een term tussen twee accolades betekent dat de term tussen de accolades *een of meer keren* moet worden ingevuld. Ook hier geldt weer dat twee herhalingen moeten worden gescheiden door een komma.
   { A } ...  Bijvoorbeeld A, A, A

Bovendien gelden de volgende afspraken:
- Hoofdletters worden gebruikt om zogenaamde sleutelwoorden ('keywords') aan te duiden en moeten letterlijk worden overgenomen.
- De woorden in kleine letters geven syntactische begrippen weer en moeten met behulp van nog volgende syntactische regels nader worden uitgewerkt.

*SQL-opdracht*

```
{ SELECT-opdracht
| INSERT-opdracht
| UPDATE-opdracht
| DELETE-opdracht }
```

*SELECT-opdracht*

```
SELECT    [ DISTINCT ] { { kolom-expressie } ... | * }
FROM      { tabelnaam [alias] } ...
[WHERE    conditie]
[GROUP BY { kolomspecificatie } ...
[HAVING   havingconditie] ]
[ORDER BY { kolomspecificatie sorteervolgorde } ... ]
```

*kolomexpressie*

```
{ tabelnaam.* | expressie | functie }
```

*kolomspecificatie*

```
{ kolomnaam | tabelnaam.kolomnaam }
```

*expressie*

```
{ numerieke-expressie | alfanumerieke-expressie }
```

*numerieke-expressie*

```
{ numerieke-constante
| kolomspecificatie
| numerieke-expressie numerieke-operator numerieke-expressie
| ( numerieke-expressie ) }
```

*alfanumerieke-expressie*

```
{ alfanumerieke-constante | kolomspecificatie }
```

*numerieke-operator*

```
{ * | / | + | - }
```

*functie*

```
{ { COUNT | SUM | AVG | MAX | MIN }
                        ( [ DISTINCT ] expressie )
| COUNT (*) }
```

*conditie*

```
simpele-conditie [ logische-operator conditie ]
```

*simpele-conditie*

```
{ nullconditie | kolomconditie | tabelconditie
| rijconditie | bijzondere-rijconditie }
```

*logische-operator*

```
{ AND | OR }
```

*nullconditie*

```
kolomspecificatie IS [NOT] NULL
```

*kolomconditie*

```
expressie relatie-operator expressie
```

*relatie-operator*

```
{ = | ~= | < | <= | > | >= }
```

*tabelconditie*

```
[ NOT ] EXISTS SELECT-opdracht
```

*rijconditie*

```
{ kolomspecificatie | constante }
        [ NOT ] IN  SELECT-opdracht
```

*bijzondere-rijconditie*

```
{ kolomspecificatie | constante }
      relatie-operator  SELECT-opdracht
```

*havingconditie*

```
functie relatie-operator { functie | constante }
```

*sorteervolgorde*

```
{ ASC | DESC }
```

*INSERT-opdracht*

```
INSERT
INTO      tabelnaam [ ( {kolomnaam} ...) ]
{ VALUES  ( { constante | NULL } ...) | SELECT-opdracht }
```

*DELETE-opdracht*

```
  DELETE
  FROM      tabelnaam
[ WHERE     conditie ]
```

*UPDATE-opdracht*

```
  UPDATE    tabelnaam
  SET       { kolomnaam = { expressie | NULL } } ...
[ WHERE     conditie ]
```

# Appendix B
# SUBSET DDL

Hieronder wordt de syntax van een DBTG schema beschreven. In die syntax komen sleutelwoorden voor die al dan niet onderstreept zijn. Een onderstreept sleutelwoord *moet* in het schema worden overgenomen; niet onderstreepte mogen worden weggelaten: ze dienen alleen ter verduidelijking voor de (menselijke) lezer van het schema. Voor de betekenis van de andere metasymbolen, zie Appendix A.

*schema*

schema entry {record entry} ... [set entry] ...

*schema entry*

SCHEMA NAME IS schemanaam .

*record entry*

record subentry [ data subentry ] ...

*set entry*

set subentry member subentry

*record subentry*

```
RECORD NAME is recordnaam
        [ KEY sleutelnaam IS { data-identifier } ...
              DUPLICATES ARE
                  { NOT ALLOWED | SYSTEM DEFAULT } ].
```

*data subentry*

```
[niveaunummer] datanaam
        { TYPE IS { DECIMAL | CHARACTER } integer-1
        | OCCURS integer-2 TIMES }.
```

*set subentry*

```
SET NAME IS setnaam
    OWNER IS { recordnaam | SYSTEM }
    ORDER FOR INSERTION IS
            { FIRST
            | LAST
            | NEXT
            | PRIOR
            | SORTED BY DEFINED KEYS } .
```

*member subentry*

```
MEMBER IS recordnaam

    INSERTION IS { AUTOMATIC | FIXED }

    RETENTION IS { MANDATORY | MANUAL | OPTIONAL }
        [ DUPLICATES ARE NOT ALLOWED
            FOR { data-identifier-1 } ... ]
        [ KEY IS { ASCENDING | DESCENDING }
            { data-identifier-2 } ...
            DUPLICATES ARE
                { NOT ALLOWED | SYSTEM DEFAULT } ]

    SET SELECTION IS THRU setnaam OWNER IDENTIFIED BY
    { APPLICATION
    | KEY sleutelnaam
        [data-identifier-3 EQUAL TO data-identifier-4]
    | SYSTEM } .
```

# Appendix C
# SUBSET DML

Voor de betekenis van de in de syntax gebruikte metasymbolen, zie Appendices A en B.

*CONTROL-opdracht*

READY [ USAGE-MODE IS
[ { EXCLUSIVE | PROTECTED | SHARED } ]
{ RETRIEVAL | UPDATE } ]

FINISH

FREE { database-sleutel-identifier
| ALL [ FROM {keeplistnaam} ... ] }

KEEP USING keeplistnaam

COMMIT

ROLLBACK

*RETRIEVAL-opdracht*

FIND record-selectie-expressie
[ RETAINING { REALM | RECORD | {setnaam} ... }
CURRENCY ]

Format 1

{ ANY | DUPLICATE }

recordnaam USING {data-identifier} ...

Format 2

DUPLICATE WITHIN setnaam

USING {data-identifier} ...

Format 3

{ NEXT | PRIOR | FIRST | LAST }

recordnaam WITHIN { setnaam | realmnaam }

Format 4

database-sleutel-identifier

Format 5

OWNER WITHIN setnaam

Format 6

recordnaam WITHIN setnaam

USING {data-identifier} ...

GET [ data-identifier ] ...

MOVE CURRENCY STATUS FOR

{ RUN-UNIT | setnaam | recordnaam }

TO data-identifier

*MODIFICATION-opdracht*

ERASE [ ALL ] [recordnaam]

MODIFY [ data-identifier ] ...

STORE recordnaam
[ RETAINING { REALM | RECORD | {setnaam} ... }
CURRENCY ]

CONNECT [recordnaam] TO { setnaam } ...

DISCONNECT [recordnaam] FROM { setnaam } ...

RECONNECT [recordnaam] WITHIN { setnaam } ...

*DATABASE condities*

setnaam { OWNER | MEMBER | TENANT }

setnaam IS [NOT] EMPTY

*STATUS indicatoren*

05 02100
02200
02300
02400

# INDEX


**A**

Afgeleide tabel 112
Alias
  van een relatie 111
Alternatieve sleutel 55
Anomalieën 36, 79
ANSI/SPARC architectuur 20
ASCENDING 157
Associatie 4-6, 62
Associatietypen
  een-op-een 5, 64
  veel-op-een 5, 64
  veel-op-veel 5, 63
Attribuut
  van een relatie 50
AUTOMATIC 160, 172
AVG 112

**B**

B-boom 220
Bachman diagram 142
Bestandsaanpak 6
Bestandsorganisatie 8
Bereik-vraag 217
Bestandbeheerder 207
Beveiliging
  van gegevens 16
Bijwerkanomalieën 79
Bijwerkoperaties
  integriteit 67
Blok 203
Boomgestructureerde index 220
Bovengelegen record 35
Boyce/Codd normaalvorm
  (BCNV) 86
Built-in functions
  zie ingebouwde functies

**C**

Candidate key
  zie kandidaatsleutel
Cartesisch produkt 108
CASCADES 68, 173
Centraal beheer
  van gegevens 14
Clustering
  zie groepering
CODASYL 137
Collisie 226
Collision
  zie collisie
COMMIT 198
Communicatie 1
Conceptuele niveau 21
Conceptuele schema 26
Conceptuele visie 26
Conceptuele/interne afbeelding
  27
CONNECT 159, 161, 186
Connection recordtype
  zie verbindingrecordtype
Connection trap
  zie verbindindingsval
Consistentie
  van gegevens 12, 37
COUNT 112
CREATE TABLE 101, 127
Current set
  van recursief settype 146
Currency indicator 177

Current record
van realm 177
van recordtype 177
van run-unit 177
van settype 177

**D**

Data definitietaal
zie data definition language
Data definition language 25, 137
Data manipulatietaal
zie data manipulation language
Data manipulation language 25, 137
Data model 28
Data opslagbeschrijvingstaal
zie data storage description language
Data storage description language 139, 229
Database key
zie database sleutel
Database sleutel 177
Database-condities 198
Data structuurdiagram 142
Data sublanguage
zie data subtaal
Data subtaal 23
Data-item
zie veld
Database administrator 12
Database model
zie data model
Database systeem 12
Databasebeheerder
zie database administrator
DB-STATUS 140, 180
DBA
zie database administrator
DBMS 14
DBTG database-ontwerp 168
DBTG set 141
DDL
zie data definition language
Deadlock 18
Declaratieve procedure 180
DELETE 68, 123, 125
Deling
van gegevens 13
Dependent record
zie ondergeschikt record
Derde normaalvorm (3NV) 85
DESCENDING 158
Determinant
van een relatie 87
Dichte index 219
DIFFERENCE 44
Directe toegang 217
DISCONNECT 159, 187
DISTINCT 104, 112
DML
zie data manipulation language 25, 137
Domain
zie domein
Domein 32, 47
DROP TABLE 126, 127
DSDL
zie data storage description language
DSL
zie data subtaal
Duplicaten
in het resultaat 104
DUPLICATES 157

**E**

Eerste normaalvorm (1NV) 78
Entiteit 3, 62, 168
Entiteit-integriteit 58
Entry
zie ingang
Equijoin 44, 106
ERASE 159, 172, 184
Existentie-test 217
EXISTS 119
Extensie
van een relatie 50

Extension
zie extensie
Extern recordtype 25
Extern schema 25
Externe niveau 21, 23
Externe visie 25
Extern/conceptuele afbeelding 28

F

FA
zie functionele afhankelijkheid
FA diagram 77
FIND 188
ANY 188
DUPLICATE 188
DUPLICATE WITHIN 191
FIRST 190
LAST 190
NEXT 190
OWNER 189
PRIOR 190
WITHIN 190
FINISH 198
FIXED 172
FIRST 158
FIXED 159
Foreign key
zie vreemde sleutel
FREE 196
FROM 102
Functioneel afhankelijkheidsdiagram 77
Functionele afhankelijkheid 76
Fysiek database-ontwerp 201
Fysieke gegevensafhankelijkheid 9
Fysiek record
zie blok
Fysiek schijfadres 203

G

Gastheertaal
zie hosttaal
Gebalanceerde boom 223
Gegevens 1
Gegevensonafhankelijkheid
fysiek 9
logisch 28
Gegevensredundantie 12, 80-84
Gegevenstransport 203
GET 182
Geïnverteerde lijst 218
Geslotenheidseigenschap van relationele systemen 102, 127
Groei-onafhankelijkheid 28
Groep
van een tabel 113
Groepering 207
Groter-dan join 108
GROUP BY 113

H

Hashing 223
HAVING 114
Heath
stelling van 83
Hiërarchische data model 34
Hosttaal 23

I

IN 116
Index 22
Index
boomgestructureerd 220
dicht 219
niet-dicht 219
primair 216
secondair 216
Index set 220
Indexering 215
Informatie 1

Ingang 216
Ingebouwde functies 111
INSERT 68, 124, 126
INSERTION 160
Integratie
  van gegevens 13
Integriteit 16
  van bijwerkoperaties 67
  van gegevens 16, 28
Intensie
  van een relatie 50
Intension
  zie intensie
Interne niveau 21, 27, 201
Interne record 27
Interne visie 27
Interrelationele functionele
    afhankelijkheid 93
Intrarelationele functionele
    afhankelijkheid 93

**J**

JOIN 42, 106

**K**

Kandidaatsleutel 55
KEEP 196
Keep list 193
Kenmerken
  niet-atomair 66
Kennis 1
KEY clausule 171
Key
  zie sleutel
KEY clausule 154
Kolom 31

**L**

LAST 158
Lege set 142
Level number
  zie niveaunummer
Logische gegevensonafhan-
    kelijkheid 28
Lossless decomposition
  zie verliesvrije decompositie
Lijst-vraag 218
Lineair zoeken 227

**M**

MANDATORY 160, 172
MANUAL 160, 172
MAX 112
Member 158
MEMBER conditie 200
MEMBERSHIP CLASS 158
MIN 112
MODIFY 68, 186
MOVE CURRENCY STATUS
    180

**N**

Natuurlijke join 44, 83, 107
Nesting
  van vragen 116
Netwerk data model 37
Netwerkvisie 37
NEXT 158
Niet-atomaire kenmerken 66,
    170
Niet-dichte index 219
Niet-sleutel attribuut 78
Niveau
  in ANSI/SPARC 20
Niveaunummer
  in COBOL 157
Normalisatie 71
NOT EXISTS 120
NULL 68
Null 58, 112, 115
NULLIFIES 68

**O**

Onafhankelijke componenten
van een relatie 94
Ondergeschikt record 35, 143
Ongebalanceerde boom 223
Ontwerpmethode
DBTG 168
relationeel 62
Opgeslagen bestand 7, 203
Opgeslagen record 6, 203
Opgeslagen veld 6, 203
Opslagschema 139
Opslagstructuur 201
OPTIONAL 172
Optimalisator 130
OPTIONAL 160
Opzoeken 102
ORDER 158
ORDER BY 104
Overflow record
zie overlooprecord
Overlappende kandidaatsleutels
86
Overlooprecord 226
Overlooptechniek
progressief 226
separaat 228
Owner 157

P

Pagina 205
Paginabeheer 209
Paginaverzameling 206
Primaire index 216
Primaire sleutel 55
PRIOR 158
PROJECT 42
Progressieve overlooptechniek
226

R

READY 198
REALM 165, 177
RECONNECT 187
RECONNECT 159
Record 7, 21-24, 139
Recursief settype 146, 170
Redundantie
van gegevens 12, 80-84
Referentie-integriteit 58
Relatie 31, 50
binair 51
cardinaliteit 51
graad 51
n-air 51
ternair 51
unair 51
Relationele algebra 44
Relationeel database-ontwerp
62
Relationele data model 31
Repeating group
zie repeterende groep
Repeterende groep 53, 170
RESTRICTED 68
RETAINING 193
RETENTION 159
Retrieval
zie opzoeken
RID 215
Rissanen
stelling van 93
ROLLBACK 198
Rotational delay 204
Run-unit 177

S

Samengesteld domein 50
Samengesteld veld 157
Schema 25, 26, 139
Schema DDL 137
Schijf 202
Schijfbeheerder 206
Search 203
Secondaire index 216
Seek 203
SELECT 42
SELECT opdracht 102

Selectieconditie 42
Separate overlooptechniek 228
Sequence set 220
Sequentiële toegang 217
SET 123, 139, 141
SET SELECTION 162
Settype
  recursief 146, 170
Singuliere set 152, 160
Sleutel 55
SORTED BY DEFINED KEYS 157
SQL 100
Statusveld 140
Storage schema
  zie opslagschema
STORE 161, 183
Structured Query Language
  zie SQL
Subschema 139, 164
Subschema DDL 137
Subvraag 116
SUM 112

## T

Tabel 31
TENANT conditie 199
Toegangstechniek 9
Transitieve afhankelijkheid 83
Tupel 31
  van een relatie 50
Tweede normaalvorm (2NV) 82

## U

Uitzonderlijke situatie 180
Uniciteitseigenschap 52
UNION 44
UPDATE 123, 124
USE FOR DB-EXCEPTION 180
User work area 140
UWA
  zie user work area

## V

Vaderbestand 228
Veld 6, 139
Verbindingrecord 39, 40
Verbindingrecordtype
  in DBTG database 148
Verbindingsval 5
Verliesvrije decompositie 83
Verliesvrije ontbinding
  zie verliesvrije decompositie
Volledig functioneel afhankelijk 77
Volledig geïnverteerd bestand 218
Vreemde sleutel 56

## W

Wederzijds onafhankelijk 78
WHERE 102
Wijzerketen 228
Wortelknooppunt
  van een index 222
Wortelrecord 35

## Z

Zoonbestand 228

In dit boek worden het relationele en het DBTG datamodel behandeld. Een datamodel bevat naast faciliteiten om data te structureren en te manipuleren (dat wil zeggen op te vragen en te veranderen), voorzieningen ten behoeve van het handhaven van de integriteit van data.
Het doel van het boek is de lezer naast kennis van begrippen uit genoemde datamodellen ook vaardigheid in het gebruik van database-systemen bij te brengen. Daarom worden er vele voorbeelden gegeven en wordt bijna ieder hoofdstuk afgesloten met een aantal (uitgewerkte) opgaven.
Na een karakterisering van databases worden de voor- en nadelen van de database-aanpak ten opzichte van de bestandsaanpak gegeven. De drie belangrijkste datamodellen van dit moment, namelijk het relationele, het hiërarchische en het netwerk-datamodel worden daarna kort besproken. Vooral de relatie tussen datastructurering en datamanipulatie komt daarbij aan de orde.
Hierna vindt concentratie op de twee genoemde datamodellen plaats en worden de manipulatietalen SQL (voor het relationele data-model) en COBOL-DML (voor het DBTG datamodel) beschreven.
Tevens wordt aandacht besteed aan het gebruik van beide datamodellen, waarbij in de relationele aanpak normalisatie natuurlijk aan de orde komt.

De inhoud van dit boek is een bewerking van hoofdstukken uit het boek *An Introduction to Database Systems,* dat door C.J. Date is geschreven.
De behandelde stof dekt de eisen voor module HB.1 van het AMBI-diploma, maar het boek is ook bedoeld voor gebruik in het hoger onderwijs, met name voor de HIO- en BIO-opleiding.

ISBN 90 6233 355 9
NUGI 852

Dit boek is geschikt voor AMBI-module HB.1

84. 50.00

ACADEMIC SERVICE ADDISON-WESLEY